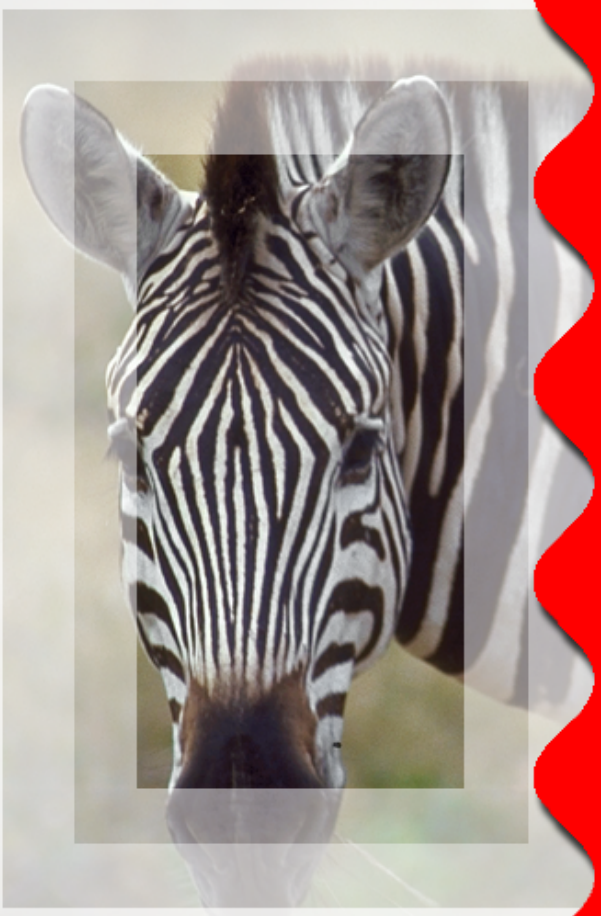
TeoImageLibrary

Programming Guide



TEO ライブラリによる画像処理プログラミング入門

Teo Image Library Programming Guide

菅谷保之

目次

第 1 章

プログラミングを始める前に

本章では，TEO 画像フォーマットや画像処理ライブラリ libteo の歴史について簡単に説明します．そして，後の章で説明する libteo によるプログラミングの準備として，ライブラリのインストール方法について解説します．

1.1 libteo とは？

libteo とは，画像フォーマットの一つである TEO フォーマット の画像を C 言語で簡単に扱うためのインターフェースを提供する画像処理ライブラリです．ライブラリはファイルの入出力や画素へのアクセス等の関数から構成され，シンプルで扱いやすくなっています．また，プラットフォームに依存しないマルチプラットフォーム環境での使用を考慮したライブラリですので，Windows や Unix などの様々な環境で使用することが可能です．

では，TEO 画像フォーマットとはどんなフォーマットでしょうか？ 近年，インターネットやディジタルカメラの普及により，コンピュータ上で扱う画像が我々の身近なものになってきています．皆さんに馴染み深い画像フォーマットといえば，BMP，JPEG，PNG，GIF などの画像フォーマットでしょう．

これらの画像フォーマットは，風景やイラストなどの視覚情報を保存するために開発されたフォーマットです．一方，画像を扱う研究分野では，画像を解析した結果を画像データに付加したり，解析データそのものを画像という形式で保存したいというニーズがありました．BMP や JPEG 等の画像フォーマットではこれらの要望を完全には実現できないため，研究者は独自の画像フォーマットを作成してきました．その一つが TEO 画像フォーマットです．

TEO 画像フォーマットおよび，画像処理ライブラリ libteo は画像処理研究のために，筑波大学 (Tsukuba)，電子技術総合研究所[*1](ETL)，岡山大学 (Okayama) の開発メンバにより 1997 年に開発されました．そして開発当時のメンバの所属先の頭文字をとって **TEO** と名付けられました．

TEO 画像フォーマットおよび，画像処理ライブラリ libteo は以下のコンセプトに基づいて開発されています．

- 汎用性および，拡張性の高い画像フォーマット
- 環境に依存せず，様々な環境で相互に利用可能
- BSD ライセンスに従うオープンソース

libteo が開発された 1997 年以降，TEO に関する様々なプログラムが開発されてきました．しかし，libteo

[*1] 現在，産業技術総合研究所

を含めてこれらのプログラムは当初からオープンソースとして開発されてきましたが，その存在は明らかにしてきませんでした．

最近では SourceForge に代表されるような WEB サイトで気軽に開発物を公開できるようになり，2003 年 11 月に TEO プロジェクトも SourceForge 上で，その開発成果を公開することになりました．

1.2　libteo インストール

プログラミングを始める前に libteo をインストールしておきましょう．libteo の最新版は https://sourceforge.jp/project/teo/ から入手することができます．平成 16 年 4 月 1 日現在の最新バージョンは 1.2.4 です．

パッケージをダウンロードし，あなたの使用している環境に合わせてインストール作業を行って下さい．

1.2.1　TAR ファイルからのインストール

TAR ファイルを圧縮したパッケージ libteo-1.2.4.tar.gz をダウンロードした場合は，次の手順に従ってインストールを行います．

まず，適当な作業ディレクトリに移り，パッケージを展開します．

```
% cd /tmp
% tar xvfz libteo-1.2.4.tar.gz
```

次に **configure** コマンドを実行して **Makefile** を作成します．ここでは，ライブラリのインストール先を **/usr/local** に設定します．これにより，ライブラリのヘッダファイルが**/usr/local/include** に，ライブラリが**/usr/local/lib** にインストールされます．

```
% cd libteo-1.2.4
% ./configure --prefix=/usr/local
```

問題なく configure スクリプトが終了したら **make** コマンドでライブラリをコンパイルし，インストールします．

```
% make
% su
[パスワード] ↵
% make install
```

以下のファイルがインストールされていれば OK です．

/usr/local/include

```
teo.h
```

teo_debug.h

/usr/local/lib

libteo.a
libteo.la
libteo.so
libteo.so.1
libteo.so.1.0.4

/usr/lib/pkgconfig

teo.pc

1.2.2 RPM ファイルからのインストール

RPM ソースファイルを libteo-1.2.4-1.src.rpm をダウンロードした場合は，次の手順に従ってインストールを行って下さい．

標準のインストール先は**/usr/local** に設定されています．インストール先に変更がなければ **rpm** コマンド (バージョン 4 以降は **rpmbuild** コマンド) を使用してバイナリパッケージを作成します．

```
% rpm --rebuild libteo-1.2.4-1.src.rpm
```

rpm のバージョンが 4 以降の環境では，

```
% rpmbuild --rebuild libteo-1.2.4-1.src.rpm
```

インストール先を変更したい場合にはスペックファイルを編集してからバイナリパッケージを作成します．

```
% rpm -ivh libteo-1.2.4-1.src.rpm
% cd $HOME/rpm/SPECS ←ディストリビューションによって異なります
% emacs libteo.spec
```

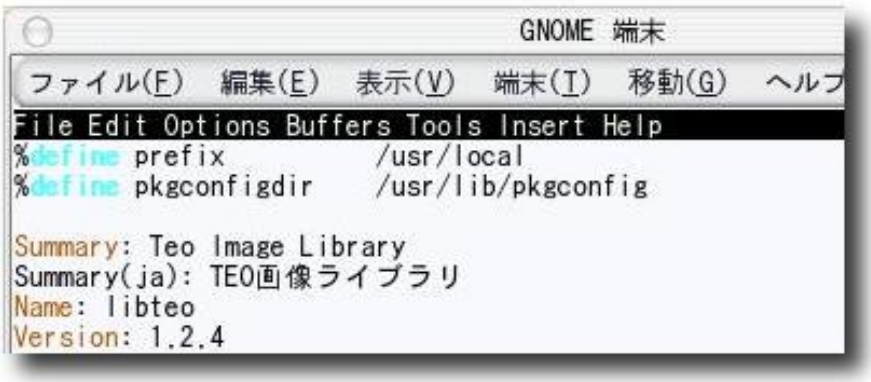


図 1.1: libteo.spec の編集画面

libteo.spec 内の prefix を希望するインストール先に編集し直します．
次のコマンドを実行してバイナリパッケージを作成します．

```
% rpm -ba libteo.spec
```

rpm のバージョンが 4 以降の環境では，

```
% rpmbuild -ba libteo.spec
```

バイナリパッケージの作成に成功すると次の 2 つのファイルが作成されます．libteo-1.2.4-1.i386.rpm にはライブラリが，libteo-1.2.4-devel-1.i386.rpm には開発に必要なヘッダファイルなどが含まれます．

$HOME/rpm/RPMS/i386
libteo-1.2.4-1.i386.rpm
libteo-devel-1.2.4-1.i386.rpm

rpm コマンドを使って libteo-1.2.4-1.i386,rpm に含まれるファイルを確認してみましょう．

```
% rpm -qpl ../RPMS/i386/libteo-devel-1.2.4-1.i386.rpm
/usr/local/include/teo.h
/usr/local/include/teo_debug.h
/usr/local/lib/libteo.la
```

最後に作成したバイナリパッケージをインストールします．

```
% rpm -ivh ../RPMS/i386/libteo-1.2.4-1.i386.rpm
% rpm -ivh ../RPMS/i386/libteo-devel-1.2.4-1.i386.rpm
```

1.3　TEO 画像ビューワ TeoEyes のインストール

TEO 画像を表示するためのビューワとして **TeoEyes** という画像ビューワが公開されています．ここでは，TeoEyes の詳細には触れずにインストール方法のみを説明します．

TeoEyes には GTK+ のバージョンに合わせて 2 系統のバージョンが開発されています．平成 1 6 年 2 月 8 日現在の最新バージョンは 2.4.7(GTK+-1.2 系) と 3.0.2(GTK+2.X 系) です．最新版は https://sourceforge.jp/project/teo/ から入手することができます．

1.3.1　バージョン 2.4.7 のインストール

バージョン 2 系列の TeoEyes をインストールするためには，以下のパッケージが必要になります．

- gtk+-1.2.x
- gdk-pixbuf-0.10.0 以降
- libteo2gdk-pixbuf-1.0.3
- teoeyes-2.4.7

gtk+，gdk-pixbuf は大抵のディストリビューションに含まれていますから，まずこれらのパッケージがインストールされている確認して下さい．

次に libteo2gdk-pixbuf のインストールを行います．

```
% cd /tmp
% tar xvfz libteo2gdk-pixbuf-1.0.3.tar.gz
% cd libteo2gdk-pixbuf
% ./configure --prefix=/usr/local
% make
% su
[パスワード] ↵
% make install
```

最後に teoeyes をコンパイルし，インストールします．

```
% tar xvfz teoeyes-2.4.7.tar.bz2
% cd teoeyes-2.4.7
% ./configure --prefix=/usr/local --datadir=/usr/local/share --with-teo-prefix=/usr
/local
% make
% su
[パスワード] ↵
% make install
```

1.3.2 バージョン 3.0.2 のインストール

バージョン 3 系列の TeoEyes をインストールするためには，以下のパッケージが必要になります．

- gtk+-2.2 以降 (ftp://ftp.gtk.org/pub/gtk/v2.2/)
- GNOME-2.2 以降 (http://ftp.gnome.org/pub/GNOME/desktop/)
  - GConf2-2.2.0 以降
  - libgnomeui-2.2.0.1 以降
  - libgnomeprint-2.2.1.1 以降
  - libgnomeprintui-2.2.1.1 以降
  - gtkhtml2-2.2.0 以降

- teoeyes-3 以降

gtk+，GNOME 関連のパッケージは予めインストールされていることを確認して下さい．
インストールの手順は今までと同様です．

```
% cd /tmp
% tar xvfz teoeyes-3.0.2.tar.bz2
% cd teoeyes-3.0.2
% ./configure --prefix=/usr/local --datadir=/usr/local/share
% make
% su
[パスワード] ↵
% make install
```

ターミナルから teoeyes と入力し，図 1.2のウィンドウが開けばインストール成功です．

```
% teoeyes
```

図 1.2: TeoEyes の起動画面

1.4 開発環境

本書では開発環境に **Fedora Core1** を使用します．C のライブラリ，コンパイラ (GCC) のバージョンを以下を示します．

```
glibc-2.3.2
gcc-3.3.2
```

以降の説明では，libteo は/usr/local 以下にインストールされているものとします．また，**pkgconfig** がインストールされていることを推奨します．

第 2 章

TEO 画像フォーマット

TEO 画像フォーマットとはどのようなフォーマットなのでしょうか？ ここでは，まず一般的な画像や画像フォーマットの特徴について説明します．次に TEO 画像フォーマットと一般的な画像フォーマットとの違いについて説明し，さらにその詳細に触れます．

2.1 一般の画像フォーマット

本節ではまず，一般的な画像の構造といくつかの代表的な画像フォーマットについて説明します．

一般に画像は図 2.1に示すように，**プレーン** (またはチャンネル) と呼ばれる複数のスクリーン (平面) から構成されます．プレーンには赤・緑・青を表現する 3 つのプレーンがあり，それらを組み合わせることで色を表現します．また α プレーンと呼ばれる透明領域を表現するプレーンを持つ画像もあります．複数のプレーンにより表現される画像平面を**フレーム**と呼びます．画像は**画素** または **pixel** と呼ばれる点から構成され (図 2.2)，各点は明るさの度合いを表す値を持ちます．この値を**画素値**，または**輝度値**と呼びます．画像フォーマットによって異なりますが，一般的には画素値は 0 から 255 の 8bit で表現されます．

一般に画素の座標は，図 2.2に示すように左上を原点として水平方向右方を X 軸，垂直方向下方を Y 軸とする座標系で表現されます．

ポイント

- 画像はプレーンと呼ばれる複数の平面で構成される
- プレーンにより構成される 1 枚の画像平面をフレームと呼ぶ
- 画像は画素と呼ばれる点で構成される
- 画素の明るさを表す値を画素値または，輝度値と呼ぶ
- 画素の座標は画像の左上を原点として，水平方向右方に X 軸，垂直方向下向に Y 軸とする座標系で表される

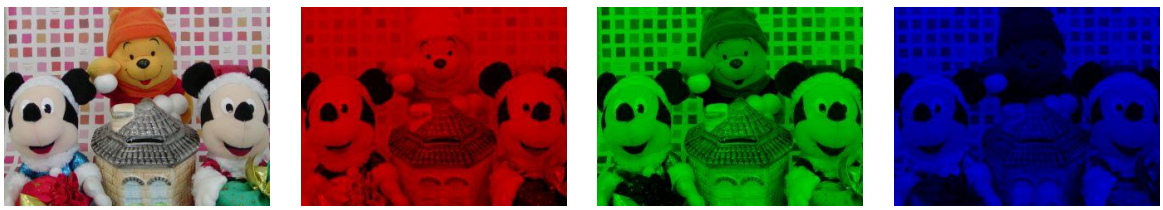


図 2.1: 赤・緑・青のプレーンによる画像の構成

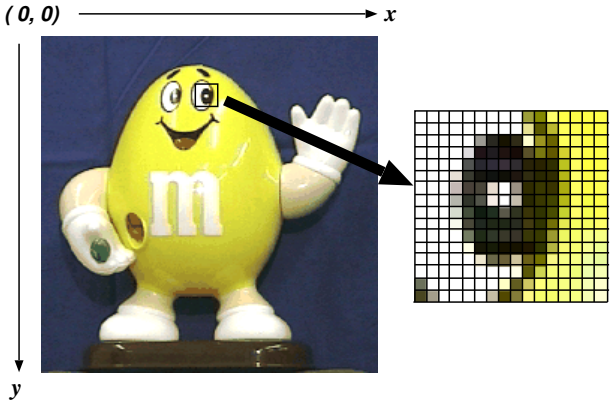


図 2.2: 画像は画素と呼ばれる点の集まり

画像フォーマットにはたくさんの種類が存在しますが，以下に代表的な画像フォーマットを示します．

これらの画像フォーマットを圧縮という観点から分類すると，圧縮系のフォーマットは JPEG，GIF，PNG，非圧縮系のフォーマットは PPM，BMP となります．圧縮系の画像フォーマットはファイルに保存するたびに各画素の値が変わってしまうため (必ずしもそうではありませんが)，画素の値が重要となる画像処理に不向きと考えられます．そういう意味で，画像処理を行う場合 UNIX 系の環境では PPM 形式の画像を，Windows 環境では BMP 形式の画像を用いることが多いようです．

それぞれのフォーマットには独自の特徴がありますが，2.1節で示した基本的な特徴は共通です．

JPEG (Joint Photographic Experts Group)　フルカラーの静止画像を圧縮するための規格．非可逆と可逆の規格が存在し，非可逆方式は DCT（離散コサイン変換）とハフマン法（もしくは算術圧縮法）を利用しています．

GIF (Graphic InterchangeFormat)　8bit カラー (256 色) またはモノクロ 256 階調に対応した圧縮形式．カラーパレットに任意の色を指定でき，必要のない色を省いて近似色で表現することで圧縮を実現しています (インデックスカラーといいます)．可逆性圧縮のある画像フォーマットです．特定の色を透明化する「透過 GIF」や，データの読み込みとともに段階的に画像を表示する「インターレース GIF」，複数の画像を連続的に表示して動画を表現する「アニメーション GIF」などの機能があります．

PNG (Portable Network Graphics)　圧縮性能が高いにも関わらず，可逆性 (圧縮してももとの画像情報は失われない) のある画像フォーマットです．インデックスカラー，グレイスケール、トゥルーカラーのイメージがサポートされ，任意で α チャンネルを加えることができます[*1]．

PPM (Portable PixMap)　主に UNIX で使用されている画像フォーマット．画像の大きさや色数などのヘッダ部と画素情報のデータ部で構成される非常にシンプルな画像フォーマットです．

BMP (Windows Bitmap)　主に Windows で使用されている画像フォーマット．Device Independent Bitmap で DIB と呼ぶ場合もあります．非圧縮，RLE 圧縮が存在します．一般の画像フォーマットが左上を原点にしているのに対し，BMP では左下を原点にしています．

[*1] http://tech.millto.net/~pngnews/kndh/PngSpec1.2/PNGcontents.html にフォーマットの詳細が記載されています

2.2 TEO 画像フォーマットって？

TEO **画像フォーマット**は PPM フォーマットを拡張して開発された画像フォーマットです．

TEO 画像フォーマットが他の画像フォーマットと大きく異なる点は，扱える画素値の型です．前節でも説明したように，一般の画像フォーマットで扱える画素値は 0 から 255 までの 8 ビットの値でした．それに対して，TEO 画像フォーマットではコンピュータで扱えるほとんどの型を画素値として使用できるように拡張されています．これには画像の解析結果など，実数値を画像として保存できる利点があります．

その他にも画像処理研究を目的として，幅広く利用できるように，以下に挙げる拡張がされています．

- **画素値の型の拡張**
  PPM では 0 から 255 までの符号なし 8bit の値しか扱えませんでした．画像を単なる絵として扱うには十分なのですが，研究では色情報だけでなく各画素の持つ様々な情報を画像の形で保存したい場合が多くあります．データを画像形式で保存する利点には，直観的なわかりやすさに加えて画像として視覚的にデータを眺めることができる点にあります．
  TEO 画像フォーマットではこのような理由から，コンピュータで扱えるほとんどのデータ型を画素値として扱えるようにしました．扱えるデータ型の詳細は第 3.2.3節を参照して下さい．
- **座標系の拡張**
  一般の画像では画像の左上を原点として，X 座標を水平方向右向きを正，Y 座標を垂直方向下向きを正とする座標系を用いています．TEO 画像フォーマットでは，座標軸の張り方は上記と同様ですが (変更も可能)，原点は任意の位置に定めることができます．
- **プレーンの拡張**
  PPM 画像フォーマットでは画像プレーン数はグレースケール画像 (PGM) で 1 枚，カラー画像で 3 枚と決まっていました．上記と同様の理由で，TEO 画像フォーマットでは，各プレーンにどのような情報を持たせるか，また何枚のプレーンを使用するかはユーザが自由に設定できるようになっています．しかし，異なるプレーンで異なるデータ型を用いることはできません．
- **フレームの拡張**
  フレームとは一枚の画像の単位のことを言います．動画ファイルでは一つのファイルで複数の画像データ (複数のフレームデータ) を保持しています．このように動画データへの対応として，TEO 画像フォーマットでも一つのファイルに複数の画像データを格納することができるような拡張がされています．
- **圧縮**
  TEO 画像フォーマットは可逆圧縮に対応しています．後の章で説明する TEO ライブラリを使用することで，ユーザはファイルが圧縮されているかどうかを気にすることなく扱うことが可能です．
- **ユーザ拡張 (User Extension)**
  TEO 画像フォーマットはファイルのヘッダ部，フッタ部を利用してユーザの自由に機能を拡張することが可能です．

2.3　フォーマットの詳細

TEO 画像フォーマットは図 2.3に示すようにヘッダ部，データ部，フッタ部で構成されます．フッタ部は省略することが可能です．図 2.2の画像ファイルは図 2.4 に示す構成になっています．１行目から４行目までがヘッダ部でそれ以降がデータ部です．この画像ファイルではフッタ部が省略されています．

2.3.1　ヘッダ部

ヘッダ部には画像サイズやプレーン数，フレーム数等の画像を情報を記述します (図 2.5)．ヘッダ部のフォーマットは次のようになります．ここで <D> はデリミタで，１個以上のスペース (0x20) または TAB(0x09)，<CR> は改行 (0x0a) を表します．[ ] 内の項目は省略可能です．

- **マジックナンバ**
  0x54，0x45，0x4f の 3 バイトのデータで，アスキーコードの’T’，’ E’，’O’ にあたります．また，バージョンナンバは’0’ または，’ 1’ が入ります．特に理由がない限り’1’ とします．
- **画像の幅，高さ，オフセット**
  画像の大きさです．オフセットは画像の左上の座標を示し，省略時は $(0, 0)$ となります．
- **画素値の型，ビット数**
  画素値の型は U/S/F のいずれかの文字で表され，それぞれ，**U は符号なし整数，S は符号あり整数，F は不動小数点数**を表します．また画素値のビット数は **1,8,16,32,64** のいずれかの値です．
- **プレーン数，フレーム数**
  画像のプレーン数，フレーム数を表します．
- **コメント**
  コメントとして扱うのは次の２通りです．
  – 行頭が#で始まる行．ただしヘッダの最初の行と最後の行にはコメントを入れてはいけません．
  – 行末の#以降．ただし他のヘッダと#の間には必ずデリミタを入れる必要があります．
  行頭が#% で始まるコメント行は特殊な意味を持ちユーザ拡張として使用することができます．

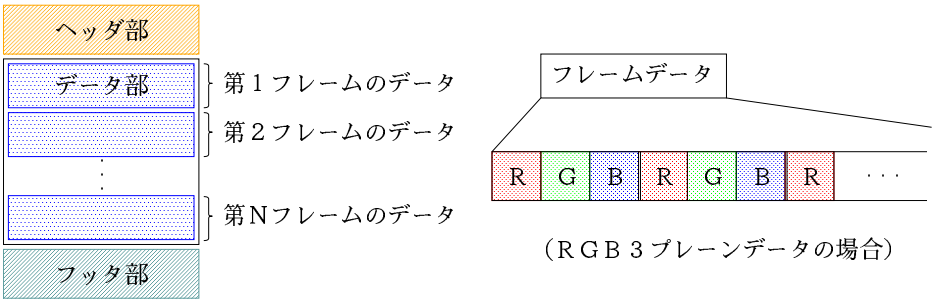


図 2.3: TEO 画像フォーマットの構造

```
TEO 1 # generated by TEO library version Version 0.4.
225 225 0 0
U 8
3 1
5;b2>g5Aj7Cl5Aj0;h2=j2:h5Bg7Di:Fo:Fo9Bl9Bl7Bo5@m5@m5@m:Er<Al7<g9Bl9Bl7@j7Cl5Aj7Cl
7Cl9Bl9Bl5@m5@m2?d<Fl7Ag4=g7@j9Ci<Fl7Di:Gl5Bg2<b0:‘5Bg5Bg2?d5Bg7Di9Ci<Fl5Aj5Aj?El
7=d9Ci:@g7=d:@g<Bi5Bg5Bg:Fo7Cl2>g4>d9Ci4?a4?a7Ag9Ci<Bi9Bl9Bl5Aj2>g:Fo5Aj0;h2=j>Ft
ENx<Eo2<b4>d:@g5;b:Ad4>d4>d5Bg5Bg7Bd7Bd:@g<Bi9Ao7@j9Bl5Bg:Gl7=d<Bi<>g<Gi9Df2=_2=_
)4V0;]?ElFLs7>a+6T2=[5<_:Ad:Ad:Ad:Ad:@g<Bi9Df9Df<Gi9Df7Bd9Df7B‘2@]5C‘9Df4?a7Bd9Df
<Cf<Cf?Fi<Bi:@g:@g<Bi7Ag7Ag7=d7>a:Ad7Ag7Ag7Di7Di?El5;b:@g7Bd4?a9Df7Bd:A‘:A‘7Bd7Bd
7Bd2@a2@a2=_7Bd:@g:@g7=d7=d:@g:@g:@g:@
                                    ...
                                                                    -.;8-B?4B?4LI0>
NL=XVGZXIZXIXVG_]NcaR^WNWQDWQD]TD_VF_VF_VFSN=SN=a\KUTAUTA\WF\WF\WF\WFdZN^XK\VI\[H
ZYFWQDZTGWPGWPG\YN_\Q\YNXUNXUNWWSYYUTYSMQSAEGO7@$+4+0A).?)2E,5H)2E
```

図 2.4: TEO ファイルの構成

マジックナンバ<D>**バージョンナンバ** [<D>**#コメント**]<CR>
[**#コメント**<CR>[**#コメント**<CR>...]]
画像の幅<D>**画像の高さ** [<D>X **座標オフセット**<D>Y **座標オフセット**][<D>**#コメント**]<CR>
[**#コメント**<CR>[**#コメント**<CR>...]]
画素値の型<D>**画素値のビット数** [<D>**#コメント**]<CR>
[**#コメント**<CR>[**#コメント**<CR>...]]
プレーン数<D>**フレーム数** [<D>**#コメント**]<CR>

図 2.5: ヘッダ部のフォーマット

2.3.2 データ部

画像データはヘッダ部の後にバイナリデータとして格納されます．データ部のサイズはヘッダ情報から次の式で計算できます．

$$\text{画像の幅} \times \text{画像の高さ} \times (\text{画素値のビット数}/8) \times \text{プレーン数} \times \text{フレーム数}\ byte \tag{2.1}$$

画素値のビット数が 1 の場合のデータ部のサイズは次のようになります．

$$\Big(\lfloor(\text{画像の幅} - 1)/8\rfloor + 1\Big) \times \text{画像の高さ} \times \text{プレーン数} \times \text{フレーム数}\ byte \tag{2.2}$$

ここで，$\lfloor\cdot\rfloor$ は床関数 (floor)．

画像が RGB の 3 プレーンからなる画像の場合，画像データは図 2.3に示すように，各点毎に RGB データを並べて格納されます．複数フレームを持つ画像の場合，各フレーム毎にまとめてデータが格納されます．

画素値のビット数が 16, 32, 64 のいずれかの場合，すなわち 1 画素のデータが複数バイトで表現される場合，TEO 画像フォーマットでは上位バイトからファイルに書き込まれます．また，画素値のビット数が 1 の場合，各画素を上位ビットから順に 8 画素ずつパックして格納されます．複数のプレーンがある場合，プレーン毎に 8 画素ずつパックされます．画像の幅が 8 の倍数でない場合は，半端なビットを無視してパックします．

2.3.3　フッタ部

フッタ部には決められたフォーマットはありません．ユーザがどんなデータを書き込んでもかまいません．

2.4　TEO 画像の圧縮

環境変数 **TEO_GZIP** が **yes** に設定されている場合，TEO 画像ファイルは自動的に圧縮されます．この圧縮は可逆圧縮ですので，圧縮により画素値の情報が変化することはありません．環境変数の詳細は 3.3を参考にして下さい．

libteo を使用することでユーザはファイルが圧縮されているかいないのかを意識することなく TEO 画像を扱うことができます．ユーザが独自に TEO ファイルの圧縮・伸長を行う場合は，ファイルの二重圧縮を防ぐためにも，**teogzip**，**teogunzip** コマンドを用いることを推奨します．

第 3 章

TEO プログラミングを始めよう

「百聞は一見にしかず」，詳しい説明をする前に簡単な例で libteo を使った画像処理プログラムを体験して，プログラムの流れを理解して下さい．ここでは，バグやエラーを起こさないためのプログラミング方針や Makefile によるコンパイルの自動化についても簡単に解説します．

3.1 とりあえず始めよう

リスト 3.1 に示したのは入力画像を出力画像にコピーするプログラムです．このリストを解説しながら，プログラムの流れを学習しましょう．

リスト 3.1 の内容は次のように分類することができます．

1–2 行目 プリプロセッサ
5–7 行目 変数定義
11–21 行目 入力画像データの読み込みと出力画像データの生成
24–31 行目 画像データのコピー
33–41 行目 画像ファイルのクローズとデータの解放

2 行目で libteo のヘッダファイル **teo.h** をインクルードしています．libteo が提供する関数を使用するためには必ずこのヘッダファイルをインクルードしなければいけません．

6–8 行目に libteo で定義された構造体，データ型が使用されています．これらの詳細については 3.2節で解説します．

11 行目で入力 TEO 画像をオープンします．TEO 画像のオープンには **TeoOpenFile** 関数を使用します．

14 行目では入力 TEO 画像と同じサイズの TEO 画像ファイルを作成しています．**TeoCreateSimilarFile** 関数はすでにオープンした TEO 画像ファイル (TEOFILE 構造体) を指定して，同じパラメータを持つ TEO 画像ファイルを作成する関数です．

17，18 行目では **TeoAllocSimilarImage** 関数を使用して，入力画像，出力画像用の画像領域をメモリ上に確保しています．

21 行目の **TeoReadFrame** 関数によりファイルから画像データを読み込み，メモリ上にデータをコピーします．TeoAllocSimilarImage を呼び出しても，メモリを確保しただけで画像データはメモリ上に読み込まれていないので注意しましょう．

24–25 行目のループは画像全体を走査するためのループです．マクロ **TeoXstart**, **TeoXend**, **TeoYstart**, **TeoYend** により，画像の四隅の座標を知ることができます．このループの中で，入力画像の各点の画素値を

取得し，出力画像の同一点に取得した画素値を書き込んでいます．画素値を取得する関数は **TeoGetPixel**，画素値を書き込む関数は **TeoPutPixel** です．画素値はプレーンごとに書き込まれていますから，26 行目のループで各プレーンについて調べています．画像のプレーン数を調べるには **TeoPlane** を使用します．

画像の走査が終了したら，**TeoWriteFrame** 関数によりメモリ上の画像処理結果を画像ファイルに書き込みます．これは TeoReadFrame と対になる関数で，この関数を呼び出して，画像データをファイルに書き込まなければ，処理結果は画像ファイルには反映されません．

最後に，38–43 行目で画像データ用のメモリ領域を解放 (**TeoFreeImage**) し，ファイルをクローズ (**TeoCloseFile**) して終了です．

実際にリスト 3.1 をエディタで入力し，copy.c という名前で保存して，以下のようにしてコンパイルしてみて下さい．

```
% gcc copy.c -o copy -I/usr/local/include -L/usr/local/lib -lteo
```

pkgconfig がインストールされている場合，次のようにオプションを指定することも可能です．

```
% gcc copy.c -o copy ‘pkg-config --cflags --libs teo‘
```

コンパイルに成功し，ディレクトリ内に copy というプログラムが作成されたら，次のようにして実行してみましょう．出力された画像を teoeyes で表示してみましょう．入力画像と同じ画像を表示されることが確認できるはずです．

```
% ./copy color.teo color-copy.teo
% teoeyes color-copy.teo
```

リスト 3.1: 画像のコピー

```
1  #include <stdio.h>
2  #include <teo.h>
3
4  int main (int argc, char *argv[]) {
5    int         x, y;
6    TEOFILE     *src_teofp, *dst_teofp;
7    TEOIMAGE    *src_img, *dst_img;
8    TEO_UINT8   val;
9
10   /* 入力ファイルのオープン */
11   src_teofp  = TeoOpenFile (argv[1]);
12
13   /* 出力ファイルの生成 */
```

```
14    dst_teofp = TeoCreateSimilarFile (argv[2], src_teofp);
15
16    /* 画像データ用のメモリを確保 */
17    src_img = TeoAllocSimilarImage (src_teofp);
18    dst_img = TeoAllocSimilarImage (dst_teofp);
19
20    /* 入力画像データをメモリに読み込む */
21    TeoReadFrame (src_teofp, src_img);
22
23    /* 各画素値をコピーする */
24    for (y = TeoYstart (src_img); y <= TeoYend (src_img); y++) {
25      for (x = TeoXstart (src_img); x <= TeoXend (src_img); x++) {
26        for (p = 0; p < TeoPlane (src_img); p++) {
27          /* 入力画像の画素値を取得 */
28          val = TeoGetPixel (src_img, x, y, p, TEO_UINT8);
29          /* 画素値を出力画像データに書き込む */
30          TeoPutPixel (dst_img, x, y, p TEO_UINT8, val);
31        }
32      }
33    }
34    /* 出力画像データを出力ファイル書き出す */
35    TeoWriteFrame (dst_teofp, dst_img);
36
37    /* 画像ファイルをクローズ */
38    TeoCloseFile (src_teofp);
39    TeoCloseFile (dst_teofp);
40
41    /* 画像データ用メモリを解放 */
42    TeoFreeImage (src_img);
43    TeoFreeImage (dst_img);
44
45    return 0;
46  }
```

3.2　libteo で定義された構造体・データ型

この節では libteo で定義されている構造体や画素値の型などについて解説します.

3.2.1　TEOFILE 構造体

TEOFILE 構造体は TEO 画像ファイルへのファイルポインタのようなものです．画像サイズやプレーン数，フレーム数等の画像情報をメンバに持つ構造体です．TEOFILE 構造体は画像データをメンバに持ちません．画像データを扱うには，TEOFILE 構造体から指定したフレームの画像データを読み込み，次に説明する TEOIMAGE 構造体へコピーしなければいけません.

TEOFILE 構造体の構造をリスト 3.2 に示します．libteo には TEOFILE 構造体のメンバへのアクセス関数 (マクロ) が用意されています.

リスト 3.2: TEOFILE 構造体

```
typedef struct {
  int  width;        /* 画像の幅 */
  int  height;       /* 画像の高さ */
  int  xoffset;      /* X 座標のオフセット値 */
  int  yoffset;      /* Y 座標のオフセット値 */
  int  type;         /* 画素値の型 */
  int  bit;          /* 画素値のビット数 */
  int  plane;        /* プレーン数 */
  int  frame;        /* フレーム数 */
  int  current;      /* 次にアクセスしようとするフレーム番号 */
  int  extc;         /* ユーザ拡張用カウンタ */
  char **extv;       /* ユーザ拡張用文字列へのポインタ */
  int  fsize;        /* 1 フレームのデータサイズ */
  int  hsize;        /* データ部の先頭へのポイント */
  FILE *fp;          /* ファイルへのポインタ */
  int  ac_type;      /* アクセスモード */
  char *filename;    /* ファイル名 */
  char *tmpfile;     /* テンポラリファイル名 */
} TEOFILE;
```

3.2.2　TEOIMAGE 構造体

TEOIMAGE 構造体の構造をリスト 3.3 に示します．TEOIMAGE 構造体は TEO 画像データをメンバに持つ構造体です．TEO 画像フォーマットはマルチフレームに対応した画像フォーマットであるため，画像ファイルへアクセスするための構造体として TEOFILE と，1 フレームの画像データへアクセスするための構

造体として TEOIMAGE の 2 つの異なる役割をする構造体が用意されているのです．

画像データにアクセスするためのマクロとして **TeoData** が用意されています．もちろん，その他のメンバにアクセスするためのマクロも用意されています．詳細については，関数リファレンスを参照して下さい．

リスト 3.3: TEOIMAGE 構造体

```
typedef struct {
  int  width;         /* 画像の幅 */
  int  height;        /* 画像の高さ */
  int  xoffset;       /* X 座標のオフセット値 */
  int  yoffset;       /* Y 座標のオフセット値 */
  int  type;          /* 画素値の型 */
  int  bit;           /* 画素値のビット数 */
  int  plane;         /* プレーン数 */
  char *data;         /* 画像データ */
  int  fsize;         /* 1 フレームのデータサイズ */
} TEOIMAGE;
```

3.2.3 画素値の型

libteo では，画素値の型として以下に示すデータ型が定義されています．このように，libteo ではコンピュータで扱える多くの型を画素値の型として使用することができるようになっています．

プラットホームに依存しないプログラムを書くために，画素値の型には必ずこれらの型を用いるようにして下さい．

TEO_BIT ... 0,1 の 2 値
TEO_UINT8 ... 8bit の符号なし整数
TEO_SINT8 ... 8bit の符号あり整数
TEO_UINT16 ... 16bit の符号なし整数
TEO_SINT16 ... 16bit の符号あり整数
TEO_UINT32 ... 32bit の符号なし整数
TEO_SINT32 ... 32bit の符号あり整数
TEO_FLOAT32 ... 32bit の浮動小数点数
TEO_FLOAT64 ... 64bit の浮動小数点数

また，画素値の型を分類するために次の値が定義されています．これは TeoCreateFile 関数や TeoAllocImage 関数の画素値の型を指定する引数として使用することができます．

TEO_UNSIGNED ... 符号なし整数
TEO_SIGNED ... 符号あり整数
TEO_FLOAT ... 浮動小数点数

3.3　環境変数

libteo ではライブラリ内で使用されるいくつかの環境変数が存在します．以下で説明する環境変数によって，TEO 画像の圧縮の有無，圧縮・伸長の際に利用するコマンドを定義しています．

- **TEO_GZIP**
  この環境変数によって生成する TEO 画像を圧縮するかどうか判定します．yes であれば圧縮，no であれば非圧縮となります．

```
% setenv TEO_GZIP yes
```

- **TEO_GZIP_COMMAND**
  圧縮の際に使用するコマンドを指定します．特に指定しない場合，ライブラリのコンパイル時に指定されたコマンドが使用されます．

```
% setenv TEO_GZIP_COMMAND ''/usr/bin/gzip -c''
```

- **TEO_GUNZIP_COMMAND**
  圧縮された TEO 画像の伸長に使用するコマンドを指定します．デフォルトのコマンドは圧縮と同様にライブラリのコンパイル時に指定されたコマンドが使用されます．

```
% setenv TEO_GUNZIP_COMMAND ''/bin/zcat''
```

- **TEO_TMP_DIR**
  圧縮・伸長のための作業用ディレクトリを指定します．特に指定しない場合，/tmp が作業用ディレクトリとなります．

```
% setenv TEO_TMP_DIR ''$HOME/tmp''
```

使用するたびにこれらの設定を行うのは面倒ですから，**.login** などのログイン時に読み込まれる設定ファイルに上記の設定を記述しておくと便利です．図 3.1は.login ファイルへの記述例です．

図 3.1: .login ファイルへの環境変数の設定

3.4 デバッグモード

libteo にはバージョン 1.1 からデバッグ支援機能として**デバッグモード** が実装されています．デバッグモードには次の 3 つのデバッグレベルがあります．

- **レベル 1**
  - TeoGetPixel，TeoPutPixel で画素座標，プレーン番号が引数に与えた画像の範囲を越えていれば警告する
  - TeoGetPixel，TeoPutPixel を実行する時に TEOIMAGE が NULL であれば警告する
  - TeoAllocImage を実行する時に画像の幅や高さが負の値なら警告する
  - TeoGetPixel，TeoPutPixel，TeoGetBit，TeoPutBit で指定した画素の型と実際の画素の型のサイズが異なっていたら警告する
- **レベル 2**
  - TeoCloseFile を実行した時に最終フレームでない (TeoWriteFrame の実行し忘れ) なら警告する
  - TeoAllocImage を実行した時に画像サイズが異常に大きい場合 (100MB 以上) に警告する
  - TeoOpenFile，TeoCreateFile でファイル名が*.teo で終ってなければ警告する
- **レベル 3**
  - その他全ての不審な挙動に対して警告する

デバッグモードでプログラムをコンパイルするには，コンパイルオプションに次のオプションを指定します．

-DTEO_DEBUG1
-DTEO_DEBUG2
-DTEO_DEBUG3
-DTEO_DEBUG_ALL

1,2,3 はデバッグレベルを表します．**-DTEO_DEBUG_ALL** オプションは全てのデバッグレベルを有効するオプションです．現在は**-DTEO_DEBUG3** と同じ意味です．

-DTEO_DEBUG_ERROR

オプションを同時に指定すると，警告を出すところを全てエラーにして停止するようになります．

バグのないクリーンなプログラムを作成するためにも，始めはデバッグモードを有効にしてプログラムを作成し，プログラムが完成した後でデバッグモードを解除してコンパイルし直すことをお奨めします．

3.5　エラー処理

本節では libteo を使って画像処理プログラムを作成する際のエラー処理について説明します．エラー処理は必ずしも必要というわけではありませんが，安全なプログラムを作成するためには重要な要素です．

ここでは，リスト 3.1 に対してエラー処理を追加しながら，どのようなエラー処理が必要になるか説明しましょう．

ソースはリスト 3.4 になります．このプログラムでは次の 6 つのエラーチェックを行っています．

1. **プログラムの引数は正しく与えられたか**
   このプログラムでは引数として入力画像ファイル名と出力画像ファイル名の 2 つの引数を与えなければいけません．そこで，プログラムの引数の数を表す変数 argc を調べて，引数の数が合わなければ USAGE を表示して終了します．
2. **入力ファイルを正常に開くことができたか**
   正常にファイルがオープンできなかった場合，TeoOpenFile は NULL を返します．その時はエラーメッセージを標準エラーに出力して終了します．
3. **入力画像の画素値の型は UINT8 か**
   このプログラムで扱う画素値の型は 8 ビットの符合なし整数としています．そこで，マクロ **TeoIsUINT8** を使って入力画像の画素値の型をチェックしています．
4. **出力ファイルを正常に作成できたか**
   正常にファイルが作成できなかった場合，TeoCreate{similar}File は NULL を返します．その時はエラーメッセージを標準エラーに出力して終了します．
5. **入力画像データ用のメモリを確保できたか**
   TeoAlloc{Similar}Image は正常にメモリが確保できなかった場合，NULL を返します．その時はエラーメッセージを標準エラーに出力して終了します．
6. **出力画像データ用のメモリを確保できたか**
   4. と同じ．

47 行目から始まる画像全体の走査では，TeoXstart，TeoYstart，TeoXend，TeoYend によって画像の範囲を決定していますので，画素座標 (x,y) が画像の範囲外になることはありません．このように，これらのマクロを使用することで，事前にエラーを回避することができます．

リスト 3.4: 画像のコピー (エラー処理の追加)

```
1  #include <stdio.h>
2  #include <teo.h>
3
4  int main (int argc, char *argv[]) {
5    int         x, y;
6    TEOFILE     *src_teofp = NULL, *dst_teofp = NULL;
7    TEOIMAGE    *src_img = NULL, *dst_img = NULL;
8    TEO_UINT8   val;
```

```
9
10  /* 引数のチェック */
11  if (argc != 3) {
12    fprintf (stderr, "Usage: %s in.teo out.teo\n", argv[0]);
13    exit (1);
14  }
15  /* 入力ファイルのオープン */
16  src_teofp  = TeoOpenFile (argv[1]);
17  if (!src_teofp) {
18    fprintf (stderr, "Could not open %s.\n", argv[1]);
19    goto settle;
20  }
21  /* 画素値の型のチェック */
22  if (!TeoIsUINT8 (src_teofp)) {
23    fprintf (stderr, "Pixel type UINT8 is only supported in this program.\n");
24    goto settle;
25  }
26  /* 出力ファイルの生成 */
27  dst_teofp = TeoCreateSimilarFile (argv[2], src_teofp);
28  if (!dst_teofp) {
29    fprintf (stderr, "Could not create %s.\n", argv[2]);
30    goto settle;
31  }
32  /* 画像データ用のメモリを確保 */
33  src_img = TeoAllocSimilarImage (src_teofp);
34  if (!src_img) {
35    fprintf (stderr, "Could not allocate enough memory.");
36    goto settle;
37  }
38  dst_img = TeoAllocSimilarImage (dst_teofp);
39  if (!dst_img) {
40    fprintf (stderr, "Could not allocate enough memory.");
41    goto settle;
42  }
43  /* 入力画像データをメモリに読み込む */
44  TeoReadFrame (src_teofp, src_img);
45
46  /* 各画素値をコピーする */
47  for (y = TeoYstart (src_img); y <= TeoYend (src_img); y++) {
48    for (x = TeoXstart (src_img); x <= TeoXend (src_img); x++) {
```

```
49        for (p = 0; p < TeoPlane (src_img); p++) {
50          /* 入力画像の画素値を取得 */
51          val = TeoGetPixel (src_img, x, y, p, TEO_UINT8);
52          /* 画素値を出力画像データに書き込む */
53          TeoPutPixel (dst_img, x, y, p TEO_UINT8, val);
54        }
55      }
56    }
57    /* 出力画像データを出力ファイル書き出す */
58    TeoWriteFrame (dst_teofp, dst_img);
59  settle:
60    /* 画像ファイルをクローズ */
61    if (src_teofp) TeoCloseFile (src_teofp);
62    if (dst_teofp) TeoCloseFile (dst_teofp);
63
64    /* 画像データ用メモリを解放 */
65    if (src_img) TeoFreeImage (src_img);
66    if (dst_img) TeoFreeImage (dst_img);
67
68    return 0;
69  }
```

3.6　コンパイルの自動化

プログラムのコンパイルを行う際に，いちいち gcc コマンドを入力するのは面倒です．このコンパイル作業を自動化してくれるのが **make** コマンドです．make コマンドは **Makefile** に記述された約束に従ってプログラムのコンパイルを行ってくれます．一度 Makefile を用意してしまえば，あとは make コマンド実行するだけで自動的にコンパイルが行われるわけです．

ここでは，Makefile の書き方についての詳細は省略し，Makefile の例を挙げ，libteo に関する部分について説明します．

リスト 3.5 が Makefile の例です．

まず，3 行目で libteo のインストールされているディレクトリを **TEODIR** というマクロ名で定義します．そして，4 行目の **CFLAGS** に libteo のヘッダファイル (teo.h) がインストールされているディレクトリを，5 行目の **LDFLAGS** にライブラリ本体がインストールされているディレクトリを指定します．**LIBS** には，libteo をリンクするために**-lteo** という記述を挿入します．

この例では，copy.c というファイルをコンパイルし，teo_copy というプログラムを作成するようにしていますが，**SRCS**，**PROGRAM** に指定する名前を変更することで，他のプログラムに使用することが可能です．

次の章から本格的に libteo を使用した画像処理プログラムについて解説していきますが，以後の説明ではプログラムのコンパイルはすべて make コマンドを用いて行うことにします．

リスト 3.5: Makefile の例

```
CC              = gcc -O2 -DTEO_DEBUG_ALL
INSTALL         = /usr/bin/install
TEODIR          = /usr/local
CFLAGS          = -I$(TEODIR)/include
LDFLAGS         = -L$(TEODIR)/lib
LIBS            = -Wall -lteo -lm
DEST            = $(HOME)/bin
SRCS            = copy.c
OBJS            = $(SRCS:.c=.o)
PROGRAM         = teo_copy
all:            $(PROGRAM)
$(PROGRAM):     $(OBJS)
                $(CC) $(OBJS) $(LDFLAGS) $(LIBS) -o $(PROGRAM)
clean:;         rm -f *.o *~ $(PROGRAM)
install:        $(PROGRAM)
                $(INSTALL) -s $(PROGRAM) $(DEST)
                strip $(DEST)/$(PROGRAM)
```

第 4 章

実践 TEO プログラミング - 構造編

本章では様々な例題を通して，TEO 画像フォーマットの理解を深めるとともに，それらをいかに扱うかについて学習します．以下に，この構造編で扱う題材とその目的を列挙しておきます．

1. **カラー画像から濃淡画像への変換**
   RGB カラーから濃淡値への変換を題材に画素値の扱いとプレーン構造について学びます．
2. **色空間の変換 (RGB 色空間から YCrCb 色空間への変換)**
   前の例題をふまえて異なる色変換を扱います．ここでは，異なる画素値の型を持つ画像の扱いについて学びます．また異なる色空間について，RGB から YCrCb への画素値の変換アルゴリズムについても同時に理解しましょう．
3. **マルチフレーム画像の扱い (画像の 2 値化)**
   画像の 2 値化という簡単な例をもとに，1 つのファイルに複数の画像データを持つマルチフレーム TEO 画像の扱いについて学びます．
4. **複数画像の扱い (動領域の検出)**
   複数枚の画像を 1 つのプログラムで読み込み処理する方法について学びます．ここではフレーム間差分により画像中の運動領域の検出を試みます．

4.1 例題１ カラー画像から濃淡画像への変換

この節では RGB カラー画像から濃淡画像への変換プログラムを作成することで，画素値の扱いとプレーンの構造について学びます．RGB カラーから濃淡値への変換は一般に次式が用いられます．

$$gray = 0.299 \times R + 0.587 \times G + 0.114 \times B \tag{4.1}$$

ここで R, G, B は画素値の RGB 値を表します．式 (4.1) で変換された濃淡値も RGB 値と同様に 0 から 255 までの値をとります．

次にプログラムの入出力について考えましょう．変換対象とする画像は RGB 画像ですから，TEO_UINT8 型の画素値を持つ 3 プレーンからなる画像ということになります．出力は濃淡画像ですから，TEO_UINT8 型の画素値を持つ 1 プレーンからなる画像です．プログラムへ与える引数は入力画像ファイル名だけにして，出力は標準出力に出力するようにすることにします．以上をまとめて，作成するプログラムの仕様を次に示します．プログラム名は teo_rgb2gray とします．

teo_rgb2gray の仕様

入力　：RGB 画素値を持つ TEO 画像 (画素値の型:TEO_UINT8, プレーン数: 3)
出力　：濃淡 TEO 画像 (画素値の型:TEO_UINT8, プレーン数:1)
動作　：入力画像の RGB 値を濃淡値に変換し，その濃淡値を画素値とする 1 プレーンの TEO 画像を標準出力に出力します．

ソースをリスト 4.1 に示します．ここでも説明の簡略化のため，本プログラムでは本質的でないエラー処理は省略しています．ソースはメイン関数と実際に色の変換を行う関数に分かれています．ごく簡単なプログラムはメイン関数に全ての処理を押し込んでしまいがちです．しかしソースを再利用して長く使用することを考えると，常にプログラムあるいはソースのモジュール化を考慮しながらプログラムを構成することは重要です．

それではソースの内容を見てみましょう (ソースを眺める前に上に示した仕様に沿って自分でプログラムを作成してみると更に理解が深まると思います)．

メイン関数でファイルの入出力を扱い，実際の変換処理は別関数が担当する構成になっています．まずメイン関数ですが，この部分は以前に扱ったものをほぼ同じ内容ですので詳しい説明は省略します．この中で注意すべきところは 52 行目です．仕様で入力画像は RGB 画像と決めていますから以下の条件をチェックして，条件に合わない画像が入力された時は処理を終了するようにしています．

- プレーンは 3 つ以上存在するか
- 画素値の型は TEO_UINT8 か

次に関数 *func_rgb2gray* ですが，引数を入力画像の TEOIMAGE に取り，処理結果を TEOIMAGE で返すようになっています．このように処理する部分だけを関数にまとめることでモジュール性が高まります．他のプログラムでも同じ処理が必要になった時には，この関数だけをコピーして再利用することができるのです．

関数の内容は次のようになっています．

出力画像データの領域確保 (14-20 **行目**)　出力画像データ用のメモリを確保します．入力画像データとサイズ，オフセット，画素値の型は同じですが，濃淡画像データですからプレーン数が 1 になっています．

画像の走査 (22-23 **行目**)　画像を左上から水平に順に走査して全ての画素を処理するためのループです．画像を走査する最も基本となる部分です．

RGB **値の取得** (24-26 **行目**)　入力画像データの第 1 プレーンが赤プレーン，第 2 プレーンが緑プレーン，第 3 プレーンが青プレーンとして (プログラムでは第 1 プレーンに対応するプレーン番号は 0 であることに注意して下さい)，TeoGetPixel によって，TEOIMAGE 構造体を介して画素座標 (x, y) の各プレーンでの画素値を獲得しています．

濃淡値の計算 (27 **行目**)　式 (4.1) に従って RGB 値から濃淡値を計算し，変数 gray に代入しています．この時，計算結果は実数になるため，代入の前で値を TEO_UINT8 にキャストしています．

画素値の書き込み (28 **行目**)　計算した濃淡値 gray を TeoPutPixel によって，出力画像データの画素座標 (x, y) の第 1 プレーンに書き込んでいます．

図 4.1に RGB 画像から変換した濃淡画像を示します．

リスト 4.1: RGB 画像から濃淡画像への変換

```
1  /* ********************************************************* rgb2gray.c *** */
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <teo.h>        /* TEO ライブラリ用のヘッダファイル */
5
6  /* RGB 画像から濃淡画像への変換関数 ************************************** */
7  static TEOIMAGE*
8  func_rgb2gray (TEOIMAGE        *src) {
9    TEOIMAGE      *dst;
10   int           row, col;
11   TEO_UINT8     r, g, b, gray;
12
13   /* 濃淡画像データの領域確保 */
14   dst = TeoAllocImage (TeoWidth   (src),
15                        TeoHeight  (src),
16                        TeoXoffset (src),
17                        TeoYoffset (src),
18                        TeoType    (src),
19                        TeoBit     (src),
20                        1);
21   /* 濃淡画像への変換 */
22   for (row = TeoYstart (src); row <= TeoYend (src); row++) {
23     for (col = TeoXstart (src); col <= TeoXend (src); col++) {
24       r = TeoGetPixel (src, col, row, 0, TEO_UINT8);
25       g = TeoGetPixel (src, col, row, 1, TEO_UINT8);
26       b = TeoGetPixel (src, col, row, 2, TEO_UINT8);
27       gray = (TEO_UINT8) (0.299 * r + 0.587 * g + 0.114 * b);
28       TeoPutPixel (dst, col ,row, 0, TEO_UINT8, gray);
29     }
30   }
31   return dst;
32 }
33
34 /* メイン関数 ************************************************************ */
35 int
36 main (int       argc,
37       char      **argv) {
38   TEOFILE       *src_teofp = NULL, *dst_teofp = NULL;
```

```
39   TEOIMAGE      *src_img = NULL, *dst_img = NULL;
40
41   /* 引数のチェック */
42   if (argc != 2) {
43     fprintf (stderr, "Usage: %s in.teo\n", argv[0]);
44     exit (1);
45   }
46   /* 入力画像の読み込み */
47   src_teofp = TeoOpenFile (argv[1]);              /* TEO ファイルのオープン */
48   src_img = TeoAllocSimilarImage (src_teofp); /* 画像データ用のメモリ確保 */
49   TeoReadFrame (src_teofp, src_img);          /* 画像データをメモリにコピー */
50
51   /* 入力画像をチェック */
52   if (TeoPlane (src_img) < 3 || !TeoIsUINT8 (src_img)) {
53     fprintf (stderr, "Wrong image type!\n");
54     goto settle;
55   }
56   /* 出力画像の生成 */
57   dst_teofp = TeoCreateFile ("-",
58                              TeoWidth   (src_img),
59                              TeoHeight  (src_img),
60                              TeoXoffset (src_img),
61                              TeoYoffset (src_img),
62                              TeoType    (src_img),
63                              TeoBit     (src_img),
64                              1,
65                              1);
66   /* 濃淡画像への変換 */
67   dst_img = func_rgb2gray (src_img);
68
69   /* 出力画像データを出力ファイルに書き出す */
70   TeoWriteFrame (dst_teofp, dst_img);
71 settle:
72   /* ファイルのクローズ */
73   if (src_teofp) TeoCloseFile (src_teofp);
74   if (dst_teofp) TeoCloseFile (dst_teofp);
75
76   /* 確保したデータの解放 */
77   if (src_img) TeoFreeImage (src_img);
78   if (dst_img) TeoFreeImage (dst_img);
```

```
79
80    return 0;
81  }
82
83  /* ************************************************ End of rgb2gray.c *** */
```

(a) 入力カラー画像　　(b) 濃淡画像

図 4.1: RGB カラーから濃淡値への変換結果

4.2　例題 2 色空間の変換 - RGB 色空間から YCrCb 色空間への変換 -

ここでは前の例題とはちょっと異なる色変換を扱ってみましょう．RGB 色空間を YCrCb 色空間へ変換してみます．

YCrCb 色空間とは，色情報を輝度信号 Y(濃淡信号) と色差信号 Cr, Cb (色信号) に分離したものです．Y は輝度を，Cr は赤の色差，Cb は青の色差を表します．NTSC の信号はこの YCrCb 信号です．

RGB 色空間から YCrCb 色空間への変換は一般に次式が用いられます．

$$\begin{cases} Y &= 0.2989 \times R + 0.5866 \times G + 0.1145 \times B \\ Cb &= -0.1687 \times R - 0.3312 \times G + 0.5000 \times B \\ Cr &= 0.5000 \times R - 0.4183 \times G - 0.0816 \times B \end{cases} \tag{4.2}$$

式 (4.2) を見てもわかるように，YCrCb 値は TEO_UINT8 の範囲では表現できないようです．そこで，変換して得られた YCrCb 値を画素値の型 TEO_FLOAT64 の画像として保存します．前回と同様にプログラムの仕様を考えてみてください．プログラム名は teo_rgb2yc とします．

teo_rgb2yc の仕様

入力　：RGB 画素値を持つ TEO 画像 (画素値の型:TEO_UINT8, プレーン数:3)
出力　：YCrCbTEO 画像 (画素値の型:TEO_FLOAT64, プレーン数:3)
動作　：入力画像の RGB 値を YCrCb 値に変換し，その YCrCb 値を TEO_FLOAT64 の画素値とする 3 プレーンの TEO 画像を標準出力に出力します．

ソースをリスト 4.2 に示します．

このプログラムもメイン関数と色変換を行う関数とに分かれています．色変換を題材にしていますから，前回のプログラムと比較して大部分が同じ内容になっています．

メイン関数の前回のソースとの変更点は，62 行目と 64 行目です．64 行目は関数の名前が違うだけです．62 行目では，出力画像の画素値の型が TEO_UINT8 から TEO_FLOAT64 に，プレーン数が 1 から 3 に変わっています．

色空間を変換する関数を見てみましょう．

変数の宣言 (12 行目)　Y，Cr，Cb の値用に TEO_FLOAT64 型の変数を宣言しています．

出力画像データの領域確保 (15-16 行目)　出力画像データ用のメモリを確保します．入力画像データとサイズ，オフセットまでが入力データと同じで，画素値の型は TEO_FLOAT64，プレーン数が 3 になっていることに注意して下さい．

画像の走査 (18-19 行目)　画像を左上から水平に順に走査して全ての画素を処理するためのループです．画像を走査する最も基本となる部分です．

RGB 値の取得 (20-22 行目)　入力画像データの第 1 プレーンが赤プレーン，第 2 プレーンが緑プレーン，第 3 プレーンが青プレーンとして (プログラムでは第 1 プレーンに対応するプレーン番号は 0 であることに注意して下さい)，TeoGetPixel によって，TEOIMAGE 構造体を介して画素座標 (x, y) の各プレーンでの画素値を獲得しています．

YCrCb 値の計算 (24-26 行目) 式 (4.2) に従って RGB 値から YCrCb 値を計算し，各変数に代入します．

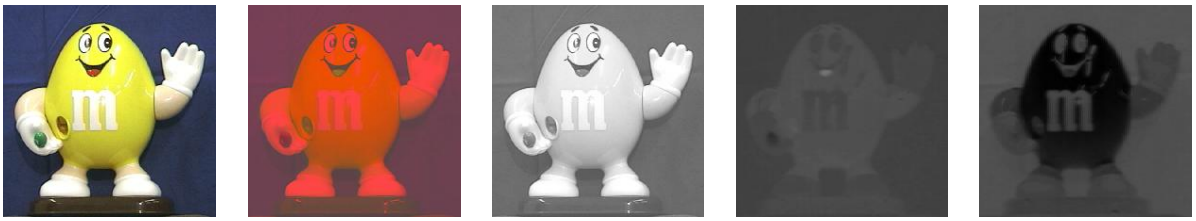
画素値の書き込み (28-30 行目) 計算した YCrCb 値を TeoPutPixel によって出力画像データの画素座標 (x, y) の各プレーンに書き込む．この時，出力画像データの画素値の型は TEO_FLOAT64 なので，TeoPutPixel に指定する画素値の型は TEO_FLOAT64 であることに注意して下さい．

リスト 4.2: RGB 画像から YCrCb 画像への変換

```
1  /* ********************************************************** rgb2yc.c *** */
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <teo.h>        /* TEO ライブラリ用のヘッダファイル */
5
6  /* RGB カラー画像を YCrCb 画像に変換する関数 ********************************** */
7  static TEOIMAGE*
8  func_rgb2yc (TEOIMAGE        *src) {
9    TEOIMAGE      *dst;
10   int           row, col;
11   TEO_UINT8     r, g, b;
12   TEO_FLOAT64   Y, Cr, Cb;
13
14   /* YCrCb 画像データの領域確保 */
15   dst = TeoAllocImage (TeoWidth (src), TeoHeight  (src),
16                        TeoXoffset (src), TeoYoffset (src), TEO_FLOAT, 64, 3);
17   /* YCrCb 画像への変換 */
18   for (row = TeoYstart (src); row <= TeoYend (src); row++) {
19     for (col = TeoXstart (src); col <= TeoXend (src); col++) {
20       r = TeoGetPixel (src, col, row, 0, TEO_UINT8);
21       g = TeoGetPixel (src, col, row, 1, TEO_UINT8);
22       b = TeoGetPixel (src, col, row, 2, TEO_UINT8);
23
24       Y  = 0.2989 * r + 0.5866 * g + 0.1145 * b;
25       Cr = 0.5000 * r - 0.4183 * g - 0.0816 * b;
26       Cb =-0.1687 * r - 0.3312 * g + 0.5000 * b;
27
28       TeoPutPixel (dst, col, row, 0, TEO_FLOAT64, Y);
29       TeoPutPixel (dst, col, row, 1, TEO_FLOAT64, Cr);
30       TeoPutPixel (dst, col, row, 2, TEO_FLOAT64, Cb);
31     }
32   }
33   return dst;
34 }
```

```
35
36 /* メイン関数 ************************************************************ */
37 int
38 main (int       argc,
39       char      **argv) {
40   TEOFILE       *src_teofp = NULL, *dst_teofp = NULL;
41   TEOIMAGE      *src_img = NULL, *dst_img = NULL;
42
43   /* 引数のチェック */
44   if (argc != 2) {
45     fprintf (stderr, "Usage: %s in.teo\n", argv[0]);
46     exit (1);
47   }
48   /* 入力画像の読み込み */
49   src_teofp = TeoOpenFile (argv[1]);           /* TEO ファイルのオープン */
50   src_img   = TeoAllocSimilarImage (src_teofp);/* 画像データ用のメモリ確保 */
51   TeoReadFrame (src_teofp, src_img);           /* 画像データをメモリにコピー */
52
53   /* 入力画像をチェック */
54   if (TeoPlane (src_img) < 3 || !TeoIsUINT8 (src_img)) {
55     fprintf (stderr, "Wrong image type!\n");
56     goto settle;
57   }
58   /* 出力画像の生成 */
59   dst_teofp = TeoCreateFile ("-",
60                              TeoWidth   (src_img), TeoHeight  (src_img),
61                              TeoXoffset (src_img), TeoYoffset (src_img),
62                              TEO_FLOAT, 64, 3, 1);
63   /* 濃淡画像への変換 */
64   dst_img = func_rgb2yc (src_img);
65
66   /* 出力画像データを出力ファイルに書き出す */
67   TeoWriteFrame (dst_teofp, dst_img);
68 settle:
69   /* ファイルのクローズ */
70   if (src_teofp) TeoCloseFile (src_teofp);
71   if (dst_teofp) TeoCloseFile (dst_teofp);
72
73   /* 確保したデータの解放 */
74   if (src_img) TeoFreeImage (src_img);
```

```
75    if (dst_img) TeoFreeImage (dst_img);
76
77    return 0;
78  }
79
80  /* ************************************************** End of rgb2yc.c *** */
```



(a) 入力カラー画像 (b) YCrCb 画像 (c) Y プレーン (d) Cr プレーン (e) Cb プレーン

図 4.2: RGB 色空間から YCrCb 色空間への変換結果

4.3　例題３ マルチフレーム画像の扱い - 画像の２値化 -

本節と次節の課題では動画データを扱いについて考えます．本節の課題で 1 つの画像ファイルに複数の画像データを含んだマルチフレーム画像の扱いについて，次節で 1 つのファイルには 1 つの画像データしか含まないが，時間的に連続したデータを持つ複数の画像ファイルの扱いについて学びます．

本節では 2 値化のプログラムを例にしてマルチフレーム画像の取り扱いを学びます．2 値化アルゴリズム自体は 1 フレームの画像データに対する処理ですので，1 フレームずつ 2 値化処理を施して，それをフレーム数だけ繰り返すという流れになります．

2 値化のアルゴリズムには最も単純な固定しきい値を用いた方法を採用します．これはあるしきい値を境に画素値がしきい値よりも小さい場合は 0，しきい値よりも大きい場合は 1 とするものです．そのため 2 値化の対象画像は濃淡画像として処理します．入力画像を濃淡画像だけに限定してもいいのですが，せっかく前の例題で RGB カラー画像を濃淡画像へ変換するプログラムを作成していますから，その時に作成した関数を利用して入力画像がカラー画像の場合は濃淡画像に変換して 2 値化することにします．

図 4.3に 2 値化プログラム teo_binary のフローチャートと仕様を示します．リスト 4.3 に teo_binary のソースを示しますが，ソースを見る前にフローチャートやプログラムの仕様を参考に自分でプログラムを書いてみて下さい．

teo_binary の仕様

入力 1 ：RGB カラー TEO 画像，または濃淡 TEO 画像
入力 2 ： 2 値化のしきい値
出力 ： 2 値画像 (画素値の型:TEO_UINT8，プレーン数:1)
動作 ：入力画像を 2 値化して，結果を標準出力に出力します．画像がマルチフレーム画像の場合も，各フレーム毎に 2 値化処理を行って結果をマルチフレームの 2 値画像として出力します．また画像がカラー画像の場合，始めに濃淡画像に変換した後で 2 値化処理を行います．

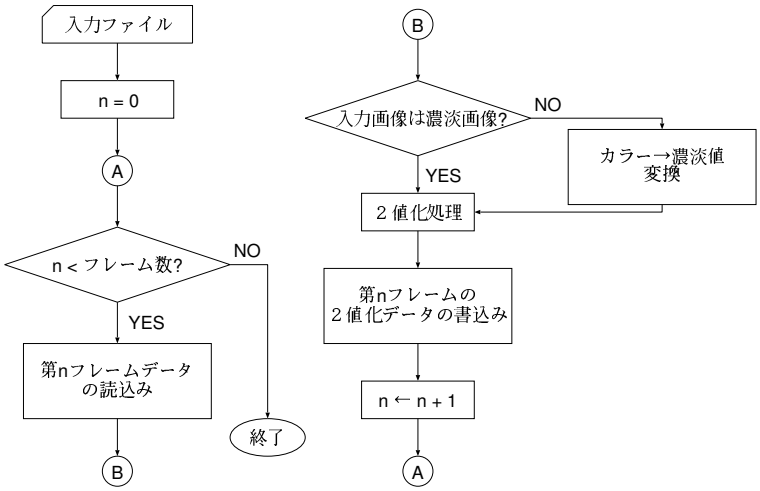


図 4.3: マルチフレーム画像対応 2 値化プログラムのフローチャート

メイン関数

フローチャートに対応する部分は 70 行目からから 80 行目になります．

フレームに対するループ (70 行目) フレーム数だけループする for 文です．

n フレーム目のデータの読み込み (71 行目) n フレーム目の画像データを読み込んで以下の処理を行う．

濃淡画像への変換 (72–74 行目) プレーン数を調べて，プレーン数が 1 でない場合 (RGB 画像と判断する)，入力画像を濃淡画像に変換します．

2 値化処理 (75 行目) *dst_img* に読み込まれている濃淡画像を入力データとして 2 値化処理を行います．結果は *dst_img* に上書きされます．

n フレーム目のデータを書き出す (78 行目) 2 値化結果をファイルに書き込みます．

TeoReadFrame と TeoWriteFrame が呼び出された後は，ファイルポインタは次のフレームのデータへ移動します．ですから，これらの関数を順番に呼び出すことで，連続的にデータにアクセスすることが可能です．

関数 *func_rgb2gray*

RGB カラーを濃淡値に変換する関数です．前回の課題では返り値は TEOIMAGE 構造体でしたが，余計なメモリ確保と解放を繰り返さないために，引数に出力データへのポインタを与えています．これによってフレームが変わっても一度確保したメモリを何度も再利用することができます．

関数 *func_binary*

関数の引数は画像データと 2 値化のしきい値です．各点の画素値を取得してしきい値と比較し，しきい値より大きい場合には出力画像の画素値を 255，そうでない場合は 0 にします．2 値化処理は各画素で処理が独立していますので 2 値化結果を入力データに上書きしています．

図 4.4に濃淡画像を入力として 2 値化した結果を，図 4.5にカラー画像を入力として 2 値化した結果を示します．どちらも 2 値化のしきい値には 128 を与えています．

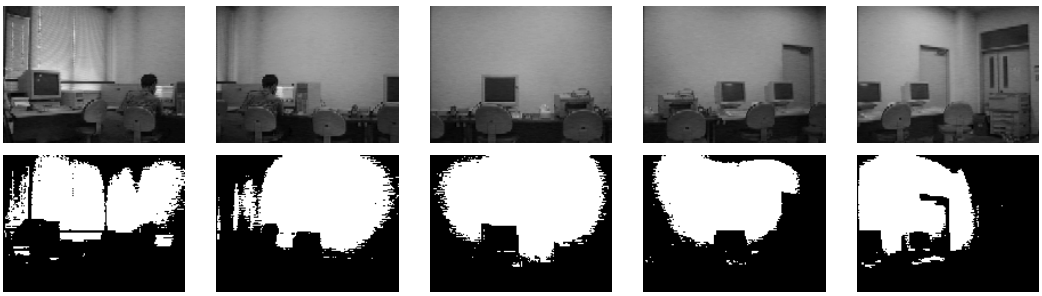


図 4.4: マルチフレーム画像の 2 値化結果 (濃淡画像) 上段:入力画像，下段:2 値画像

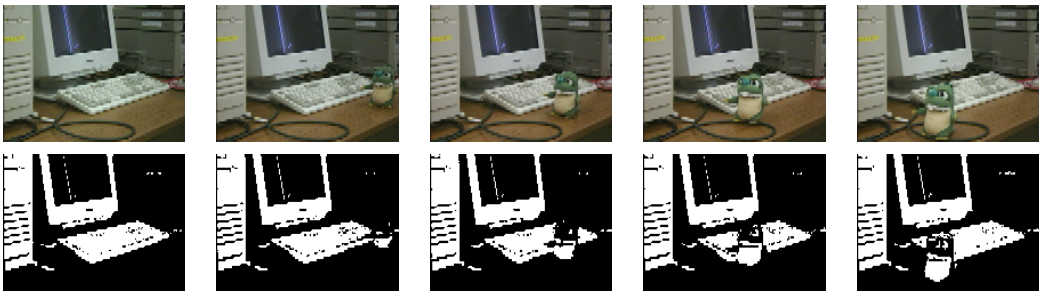


図 4.5: マルチフレーム画像の 2 値化結果 (カラー画像) 上段:入力画像，下段:2 値画像

リスト 4.3: 画像の２値化

```
/* ********************************************************** binary.c *** */
#include <stdio.h>
#include <stdlib.h>
#include <teo.h>        /* TEO ライブラリ用のヘッダファイル */

/* RGB 画像を濃淡画像に変換する関数 ************************************** */
static void
func_rgb2gray (TEOIMAGE        *src,
               TEOIMAGE        *dst) {
  int           row, col;
  TEO_UINT8     r, g, b, gray;

  for (row = TeoYstart (src); row <= TeoYend (src); row++) {
    for (col = TeoXstart (src); col <= TeoXend (src); col++) {
      r = TeoGetPixel (src, col, row, 0, TEO_UINT8);
      g = TeoGetPixel (src, col, row, 1, TEO_UINT8);
      b = TeoGetPixel (src, col, row, 2, TEO_UINT8);
      gray = (TEO_UINT8) (0.299 * r + 0.587 * g + 0.114 * b);
      TeoPutPixel (dst, col ,row, 0, TEO_UINT8, gray);
    }
  }
}

/* 画像を２値化する関数 ************************************************** */
static void
func_binary (TEOIMAGE   *src,
             TEO_UINT8  threshold) {
  int           row, col;
  TEO_UINT8     val;

  for (row = TeoYstart (src); row <= TeoYend (src); row++) {
    for (col = TeoXstart (src); col <= TeoXend (src); col++) {
      val = TeoGetPixel (src, col, row, 0, TEO_UINT8);
      val = (val > threshold) ? 255 : 0;
      TeoPutPixel (src, col, row, 0, TEO_UINT8, val);
    }
  }
}
```

```
39
40 /* メイン関数 ************************************************************ */
41 int
42 main (int       argc,
43       char      **argv) {
44   TEOFILE       *src_teofp = NULL, *dst_teofp = NULL;
45   TEOIMAGE      *src_img = NULL, *dst_img = NULL;
46   TEO_UINT8     threshold;
47   int           n;
48
49   /* 引数のチェック */
50   if (argc != 3) {
51     fprintf (stderr, "Usage: %s in.teo #threshold\n", argv[0]);
52     exit (1);
53   }
54   /* 入力画像の読み込み */
55   src_teofp = TeoOpenFile (argv[1]);          /* TEO ファイルのオープン */
56   src_img = TeoAllocSimilarImage (src_teofp); /* 画像データ用のメモリ確保 */
57
58   /* 2 値化の閾値の獲得 */
59   threshold = (TEO_UINT8) atoi (argv[2]);
60
61   /* 出力画像の生成 */
62   dst_teofp = TeoCreateFile ("-",
63                              TeoWidth   (src_img), TeoHeight  (src_img),
64                              TeoXoffset (src_img), TeoYoffset (src_img),
65                              TEO_UNSIGNED, 8, 1, TeoFrame (src_teofp));
66   /* 出力画像データ用メモリの確保 */
67   dst_img = TeoAllocSimilarImage (dst_teofp);
68
69   /* メインループ - 画像の 2 値化 */
70   for (n = 0; n < TeoFrame (src_teofp); n++) {
71     TeoReadFrame (src_teofp, src_img);
72     if (TeoPlane (src_img) != 1) {            /* RGB 画像の場合，濃淡画像に変換 */
73       func_rgb2gray (src_img, dst_img);
74     }
75     func_binary (dst_img, threshold);         /* 2 値化 */
76
77     /* 出力画像データを出力ファイルに書き出す */
78     TeoWriteFrame (dst_teofp, dst_img);
```

```
79    }
80    /* ファイルのクローズ */
81    if (src_teofp) TeoCloseFile (src_teofp);
82    if (dst_teofp) TeoCloseFile (dst_teofp);
83
84    /* 確保したデータの解放 */
85    if (src_img) TeoFreeImage (src_img);
86    if (dst_img) TeoFreeImage (dst_img);
87
88    return 0;
89  }
90
91  /* ************************************************* End of rgb2gray.c *** */
```

4.4 例題４ 複数画像の扱い - 運動領域の抽出 -

次に複数ファイルから構成される動画データから運動領域を検出するプログラムを作成してみましょう．この課題を通して，連続した複数ファイルの扱いを学びます．課題も少し複雑ですがしっかり理解して下さい．

まず運動領域検出のアルゴリズムについて説明しましょう．入力データはカメラ位置を固定して運動物体を撮影した動画データとします (図 4.6上段)．この時，背景は動きませんのでフレーム間で差分 (これはフレーム間差分と呼ばれる手法です) を取ると動きのある領域で差が大きくなります (図 4.6 中段)．両フレーム中で変化のあった領域で差分値が大きくなりますから，差分画像 1 枚からは画像中の運動物体のみを検出することができません．そこで，更に隣り合ったフレーム間で差分画像を生成し，差分画像間の共通部分を取ることで中間フレーム中の運動領域を検出します (図 4.6下段)．

次に複数ファイルの扱いついて解説します．必要な画像ファイル名をプログラムの引数に全て与える方法もありますが，少しスマートな方法を説明します．ここでは画像ファイル名が，[**ヘッダ**][**フレーム番号**].**teo** という形式であるとします．

例えば，入力ファイルが $image00.teo$, $image01.teo, \cdots, image09.teo$ の時，入力ファイルのフォーマットを $image\%02d.teo$ とすると，$n = 0, \cdots, 9$ と変化させることで入力ファイル名を自動的に生成することができます．フォーマットから連続したファイル名を生成するソースは次のようになります．

リスト 4.4: 連続ファイル名の生成

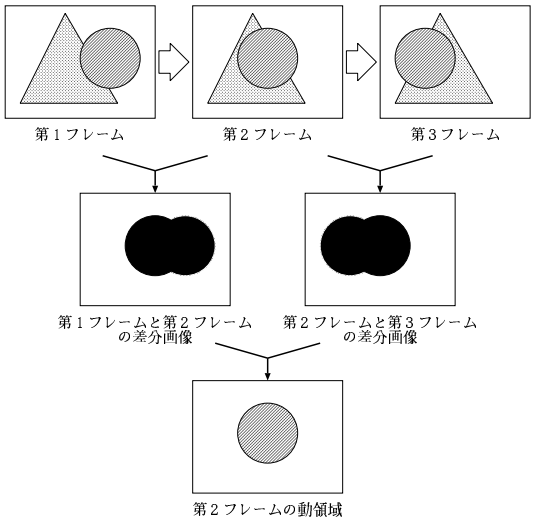


図 4.6: 動領域検出のアルゴリズム

```
1  int    n;
2  char   name[1024];
3  for (n = 0; n < 10; n++) {
4    sprintf (name, "image%02d.teo", n);
5    printf ("%s\n", name);
6  }
```

teo_detect_motion の仕様

入力 1 ： 入力画像フォーマット
入力 2 ： フレーム数
入力 3 ： 出力画像フォーマット
出力 ： 検出した運動領域画像
動作 ： フレーム間差分によって各フレームの運動領域を検出します．入力画像と出力画像は 1 フレームからなる複数のファイルとし，ファイル名はフォーマットを与えてプログラム中で自動生成するものとします．

いつものようにプログラムの仕様を載せておきました．ソースを眺める前に自分でチャレンジして見て下さい．

リスト 4.5 のソースについて以下にまとめます．

メイン関数

引数のチェック (64-67 **行目**)　今回のプログラムの引数は入力ファイルフォーマット，フレーム数，出力ファイルフォーマットの 3 つです．
フレーム数の取得 (69 **行目**)　読み込む画像数をプログラムの引数から取得します．
入力画像データの領域確保 (72-79 **行目**)　フレーム数分の画像データ用のメモリを確保して画像データを読み込みます．ファイル名は先に説明した方法を用いています．
出力画像，差分画像データの領域確保 (81-87 **行目**)　出力画像，差分画像用のメモリを確保します．差分画像は 2 つのフレーム間の差分により生成されるため，その数は”フレーム数-1”となります
差分画像の生成 (89-91 **行目**)　$func\_sub\_image$ 関数により，各フレーム間の差分画像を生成します．
運動領域の抽出 (93-103 **行目**)　隣り合った差分画像を比較して運動領域を抽出します．入力ファイル名と同じ方法で出力ファイル名を生成し，抽出結果を書き出します．
後始末 (105-109 **行目**)　いつもの後始末です．

関数 $func\_sub\_image$
2 画像間の差分をとる関数です．差分値がしきい値 $threshold$ より大きい場合には画素値を 1，そうでない場合は 0 とします．このプログラムでは $threshold = 8$ としています．

関数 $func\_detect\_motion$

２つの差分画像を比較して，両方の画像で画素値が 1 ならば運動領域と判断し，カラー画像の画素値をコピーします．それ以外の領域は白色 (画素値 255) としています．

図 4.7の上段が入力画像列です．下段はフレーム間差分による運動物体の抽出結果です．

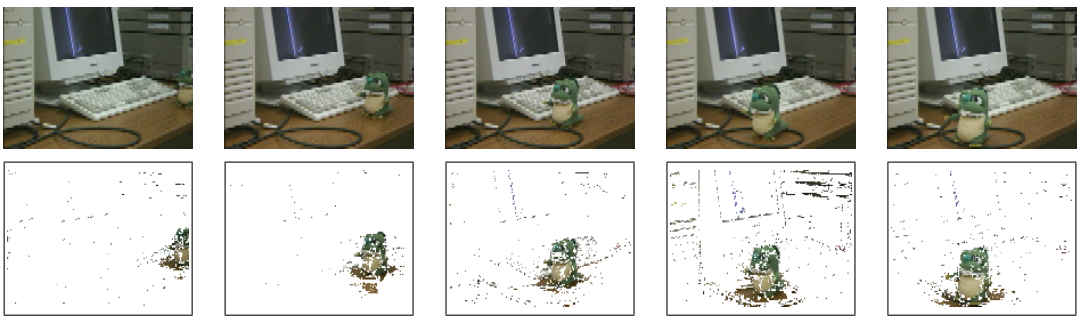


図 4.7: 動領域の検出結果 上段:入力画像，下段:検出された動領域

リスト 4.5: フレーム差分による運動領域の抽出

```
1  /* ********************************************************* subtract.c *** */
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <teo.h>        /* TEO ライブラリ用のヘッダファイル */
5
6  /* ************************************************************************ */
7  static void
8  func_and_image (TEOIMAGE        *src1,
9                  TEOIMAGE        *src2,
10                 TEOIMAGE        *color,
11                 TEOIMAGE        *dst) {
12   TEO_UINT8        val1, val2;
13   int                row, col, p;
14
15   for (row = TeoYstart (src1); row <= TeoYend (src1); row++) {
16     for (col = TeoXstart (src1); col <= TeoXend (src1); col++) {
17       val1 = TeoGetPixel (src1, col, row, 0, TEO_UINT8);
18       val2 = TeoGetPixel (src2, col, row, 0, TEO_UINT8);
19       if (val1 && val2) {
20         for (p = 0; p < TeoPlane (color); p++) {
21           TeoPutPixel (dst, col ,row, p, TEO_UINT8,
22                        TeoGetPixel (color, col, row, p, TEO_UINT8));
23         }
24       } else {
25         for (p = 0; p < TeoPlane (color); p++) {
26           TeoPutPixel (dst, col ,row, p, TEO_UINT8, 255);
27         }
28       }
29     }
30   }
31 }
32
33 /* ************************************************************************ */
34 static void
35 func_sub_image (TEOIMAGE        *src1,
36                 TEOIMAGE        *src2,
37                 TEOIMAGE        *dst) {
38   TEO_UINT8        val1, val2, threshold = 8;
```

```
39   int                row, col;
40
41   for (row = TeoYstart (src1); row <= TeoYend (src1); row++) {
42     for (col = TeoXstart (src1); col <= TeoXend (src1); col++) {
43       val1 = TeoGetPixel (src1, col, row, 0, TEO_UINT8);
44       val2 = TeoGetPixel (src2, col, row, 0, TEO_UINT8);
45       if (abs (val1 - val2) > threshold) {
46         TeoPutPixel (dst, col ,row, 0, TEO_UINT8, 1);
47       } else {
48         TeoPutPixel (dst, col ,row, 0, TEO_UINT8, 0);
49       }
50     }
51   }
52 }
53
54 /* メイン関数 ************************************************************ */
55 int
56 main (int        argc,
57       char        **argv) {
58   TEOFILE        *src_teofp = NULL, *dst_teofp = NULL;
59   TEOIMAGE        **src_img = NULL, **sub_img = NULL, *dst_img = NULL;
60   int                n, frame;
61   char                name[1024];
62
63   /* 引数のチェック */
64   if (argc != 4) {
65     fprintf (stderr, "Usage: %s in_header #frame out_header\n", argv[0]);
66     exit (1);
67   }
68   /* フレーム数の獲得 */
69   frame = atoi (argv[2]);
70
71   /* 入力画像の読み込み */
72   src_img = (TEOIMAGE **) malloc (sizeof (TEOIMAGE *) * frame);
73   for (n = 0; n < frame; n++) {
74     sprintf (name, argv[1], n);
75     src_teofp = TeoOpenFile (name);
76     src_img[n] = TeoAllocSimilarImage (src_teofp);
77     TeoReadFrame (src_teofp, src_img[n]);
78     TeoCloseFile (src_teofp);
```

```
79    }
80    /* 出力画像用メモリの確保 */
81    dst_img = TeoAllocSimilarImage (src_img[0]);
82
83    /* 差分画像用メモリの確保 */
84    sub_img = (TEOIMAGE **) malloc (sizeof (TEOIMAGE *) * (frame - 1));
85    for (n = 0; n < frame - 1; n++) {
86      sub_img[n] = TeoAllocSimilarImage (src_img[n]);
87    }
88    /* 隣合ったフレーム間の差分画像の生成 */
89    for (n = 0; n < frame - 1; n++) {
90      func_sub_image (src_img[n+1], src_img[n], sub_img[n]);
91    }
92    /* 差分画像間の AND 画像を生成 */
93    for (n = 1; n < frame - 1; n++) {
94      func_and_image (sub_img[n-1], sub_img[n], src_img[n], dst_img);
95      sprintf (name, argv[3], n);
96      dst_teofp = TeoCreateFile (name,
97                                 TeoWidth   (dst_img), TeoHeight  (dst_img),
98                                 TeoXoffset (dst_img), TeoYoffset (dst_img),
99                                 TeoType (dst_img), TeoBit (dst_img),
100                                TeoPlane (dst_img), 1);
101     TeoWriteFrame (dst_teofp, dst_img);
102     TeoCloseFile (dst_teofp);
103   }
104   /* 確保したデータの解放 */
105   for (n = 0; n < frame; n++)   TeoFreeImage (src_img[n]);
106   for (n = 0; n < frame-1; n++) TeoFreeImage (sub_img[n]);
107   free (src_img);
108   free (sub_img);
109   TeoFreeImage (dst_img);
110
111   return 0;
112 }
113
114 /* ********************************************** End of subtract.c *** */
```

第 5 章

実践 TEO プログラミング - アルゴリズム編

前章「**構造編**」では TEO 画像の構造を理解することを目的としていました．本章「**アルゴリズム編**」では，いくつかの実用的な画像処理アルゴリズムを例題にあげて，それらをどのように実装するかを解説します．

本章では，画像処理で頻繁に用いられるフィルタリングや画像の拡大・縮小の方法について扱います．この章の最後の課題では，今まで学習したことを応用して，パノラマ画像の生成を行います．なお本章では，ポイントとなる部分のソースのみ掲載しました．

本章で扱う課題の次のようになっています．

1. **ラプラシアンフィルタ**
2. **ガウシアンフィルタ**
3. **画像の拡大・縮小**
4. **画素値の内挿**
5. **パノラマ画像の生成**

5.1 例題 5 ラプラシアンフィルタ

ラプラシンフィルタは画像の鮮鋭化 やエッジ検出に使用されるフィルタです．ラプラシアンとは 2 次微分の意味です．画像の 2 次微分は次のように行います．

画像は離散的な画素の集まりですので，画像の微分は差分によって計算します．

今，図 5.1中の B を処理対象の画素とします．両隣の画素 A, C との差分を $f_x(B-A), f_x(C-B)$ とすると，これらは次式で表すことができます．ただし $I(x,y)$ は点 (x,y) での画素値を表します．

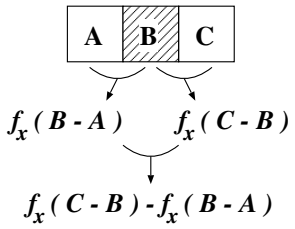
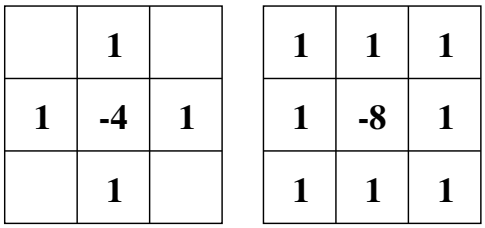


図 5.1: 画像の 2 次微分



(a)　　(b)

図 5.2: (a) 4 近傍のラプラシアンフィルタ (b) 8 近傍のラプラシアンフィルタ

$$f_x(B-A) = I(x,y) - I(x-1,y) \tag{5.1}$$

$$f_x(C-B) = I(x+1,y) - I(x,y) \tag{5.2}$$

画素 B に関する 2 次微分は，これらの 1 次微分値の差分として計算することができます (式 (5.3))．

$$\begin{aligned} f_{xx}(x,y) &= f_x(B-A) - f_x(C-B) \\ &= I(x-1,y) + I(x+1,y) - 2I(x,y) \end{aligned} \tag{5.3}$$

垂直方向の微分も同時に考えると，画素 B に関する 2 次微分は以下の式で与えられます．

$$\nabla^2 f(x,y) = I(x-1,y) + I(x+1,y) + I(x,y-1) + I(x,y+1) - 4I(x,y) \tag{5.4}$$

この式の各画素の係数をその画素の配置に合わせて表現したのが図 5.2(a) です．5.2(b) は更に斜め方向の微分も考慮したものです．これらをマスク やカーネルなどと呼びます．

ラプラシアンフィルタに限らず，フィルタリングでは処理可能な画像の範囲を考慮することが重要です．今回のラプラシンフィルタでは，隣り合った画素が存在しない画像の端の画素は処理することができません．ですから，各画素にアクセスするためのループの初期値と最終値に注意しなければいけません．

以下に示すのは，図 5.2(b) に示した 8 近傍のラプラシアンフィルタの例です．ここではカーネルサイズが 3×3 とわかっているため，画像の縁 1 画素は処理の対象から除外しています．

リスト 5.1: 8 近傍のラプラシアンフィルタの例

```
1 TEOIMAGE*
2 func_laplacian  (TEOIMAGE        *src) {
3   TEOIMAGE        *dst;
4   TEO_FLOAT64        sum;
5   int                row, col;
6
7   dst = TeoAllocImage (TeoWidth   (src), TeoHeight  (src),
8                        TeoXoffset (src), TeoYoffset (src),
```

```
                        TEO_FLOAT, 64, 1);

  for (row = TeoYstart (src) + 1; row <= TeoYend (src) - 1; row++) {
    for (col = TeoXstart (src) + 1; col <= TeoXend (src) - 1; col++) {
      sum  = (TEO_FLOAT64) TeoGetPixel (src, col-1, row-1, 0, TEO_UINT8);
      sum += (TEO_FLOAT64) TeoGetPixel (src, col  , row-1, 0, TEO_UINT8);
      sum += (TEO_FLOAT64) TeoGetPixel (src, col+1, row-1, 0, TEO_UINT8);
      sum += (TEO_FLOAT64) TeoGetPixel (src, col-1, row  , 0, TEO_UINT8);
      sum += (TEO_FLOAT64) TeoGetPixel (src, col+1, row  , 0, TEO_UINT8);
      sum += (TEO_FLOAT64) TeoGetPixel (src, col-1, row+1, 0, TEO_UINT8);
      sum += (TEO_FLOAT64) TeoGetPixel (src, col  , row+1, 0, TEO_UINT8);
      sum += (TEO_FLOAT64) TeoGetPixel (src, col+1, row+1, 0, TEO_UINT8);
      sum -= (TEO_FLOAT64) TeoGetPixel (src, col  , row  , 0, TEO_UINT8) * 8;
      TeoPutPixel (dst, col ,row, 0, TEO_FLOAT64, sum);
    }
  }
  return dst;
}
```

5.2　例題 6 ガウシアンフィルタ

ガウシアンフィルタもラプラシアンフィルタと同様よく知られたフィルタで，画像を平滑化する平滑化フィルタ の一つです．

一番単純な平滑化は周囲の画素値の平均値を取ることです．これに対してガウシアンフィルタは画素の空間的配置を考慮して，対象画素に近い画素に大きな重みを，対象画素から遠い画素には小さい重みを付けた加重平均を取ります．この重み付けにガウス関数を採用していることからガウシアンフィルタと呼ばれます．

図 5.3に 2 次元のガウス関数 $w(x, y)$ を示します．標準偏差 σ の値によって平滑化の度合を変化させることができます．

ガウシアンフィルタは式 (5.5) によって計算することができます．

$$I'(x,y) = \frac{1}{C} \sum_{k=-\sigma}^{\sigma} \sum_{l=-\sigma}^{\sigma} w(k,l) \times I(x+k, y+l) \tag{5.5}$$

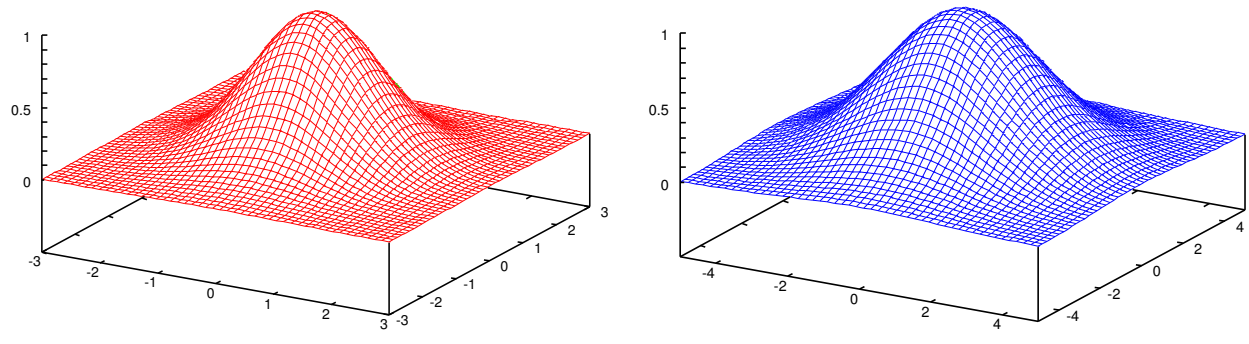
ここで，C は次の式で定める定数です．

$$C = \sum_{k=-\sigma}^{\sigma} \sum_{l=-\sigma}^{\sigma} w(k,l) \tag{5.6}$$

リスト 5.2にガウシアンフィルタの関数を示します．

この関数では，画像を走査する前にカーネルの計算を行っています．こうすることで画素ごとにガウス関数の値を計算する必要がなくなり，計算コストを削減することができます．

図 5.4にラプラシアンフィルタとガウシアンフィルタの適用結果を示します．



(a)　(b)

図 5.3: 2 次元のガウス関数 $w(x, y)$ (a) $\sigma = 1$ (b) $\sigma = 2$

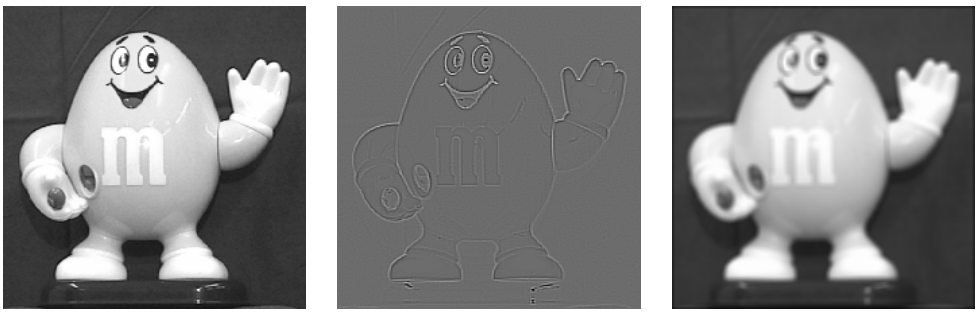
リスト 5.2: ガウシアンフィルタ

```
/* ********************************************************************** */
TEOIMAGE*
func_gaussian (TEOIMAGE *src,
               int      sigma) {
  TEOIMAGE      *dst;
  double        **gauss;
  double        sum = 0.0;
  int           kernel_size, x, y, row, col, p;

  /* 出力画像の領域確保 */
  dst = TeoAllocImage (TeoWidth   (src), TeoHeight  (src),
                       TeoXoffset (src), TeoYoffset (src),
                       TEO_UNSIGNED, 8, TeoPlane (src));

  /* ガウス関数の値を格納する領域の確保 */
  kernel_size = sigma * 2 + 1;
  gauss = (double **) malloc (sizeof (double *) * kernel_size) + sigma;
  for (y = -sigma; y <= sigma; y++) {
    gauss[y] = (double *) malloc (sizeof (double) * kernel_size) + sigma;
    for (x = -sigma; x <= sigma; x++) {
      gauss[y][x] = exp (-(x * x + y * y) / (2.0 * sigma * sigma));
      sum += gauss[y][x];
    }
  }
  /* カーネルの総和を 1 に正規化 */
  for (y = -sigma; y <= sigma; y++) {
    for (x = -sigma; x <= sigma; x++) {
      gauss[y][x] /= sum;
    }
  }
  /* 入力画像にガウシアンフィルタを適用する */
  for (row = TeoYstart (src) + sigma; row <= TeoYend (src) - sigma; row++) {
    for (col = TeoXstart (src) + sigma; col <= TeoXend (src) - sigma; col++) {
      for (p = 0; p < TeoPlane (src); p++) {
        sum = 0.0;
        for (y = -sigma; y <= sigma; y++) {
          for (x = -sigma; x <= sigma; x++) {
            sum += gauss[y][x] * TeoGetPixel (src, col+x, row+y, p, TEO_UINT8);
```

```
39            }
40          }
41          TeoPutPixel (dst, col, row, p, TEO_UINT8, (TEO_UINT8) sum);
42        }
43      }
44    }
45    /* カーネルの領域解放 */
46    for (y = -sigma; y <= sigma; y++) {
47      gauss[y] -= sigma;
48      free (gauss[y]);
49    }
50    gauss -= sigma;
51    free (gauss);
52
53    return dst;
54  }
```



(a)　(b)　(c)

図 5.4: (a) 入力濃淡画像 (b) ラプラシアンフィルタ (c) ガウシアンフィルタ

5.3 例題 7 画像の拡大・縮小

ここでは拡大・縮小による画像の変換について扱います．画像変換の中では単純なものですが，これができればほとんどの変換は可能になるでしょう．

まず画像を 2 倍に拡大することを考えましょう．まず誰もが考える方法は，入力画像の画素座標を 2 倍した位置に画素値を書き込む方法です．つまり座標の変換は次の式で与えられます．ただし (x, y) は入力画像の画素座標，(x', y') は出力画像の画素座標を表すことにします．

$$x' = 2x \tag{5.7}$$

$$y' = 2y \tag{5.8}$$

この方法で画像を拡大した結果を図 5.5に示します．縦横に黒い線が入っています．これは座標の変換に問題があるからです．例えば，点 $(i, 0), i = 0, \cdots, N$ の変換後の座標は点 $(2i, 0)$ となります．これでは点 $(2i + 1, 0)$ の画素値が計算されません．このような点が黒い線となって現れたのです．このようなことを防ぐためには，入力画像の画素座標から出力画像の画素座標を計算するのではなく，出力画像の画素座標から入力画像の画素座標を計算すればよいのです．すなわち座標の変換式は次のようになります．

$$x = x'/2 \tag{5.9}$$

$$y = y'/2 \tag{5.10}$$

このように出力画像を中心に考えれば，変換後の画像に情報の欠落した部分がなくなるわけです．拡大・縮小に関わらず入力画像から出力画像への変換式が与えられた時は，この変換の逆変換を求めて出力画像上の点の入力画像での座標および画素値を計算します．

リスト 5.3 に拡大・縮小関数を示します．計算した入力画像の座標は一般には実数になります．このソースでは計算結果を **int** 型にキャストすることで実数値を整数値に変換して，対応する画素の値を取得しています．

リスト 5.3: 画像の拡大・縮小

```
1  TEOIMAGE*
2  func_scaling (TEOIMAGE *src,
3                double   xscale,
4                double   yscale) {
5    TEOIMAGE      *dst;
6    int           p, row, col, in_row, in_col;
7
8    /* 出力画像データの領域確保 */
9    dst = TeoAllocImage ((int) (TeoWidth  (src) * xscale),
10                        (int) (TeoHeight (src) * yscale),
11                        TeoXoffset (src), TeoYoffset (src),
12                        TEO_UNSIGNED, 8, TeoPlane (src));
```

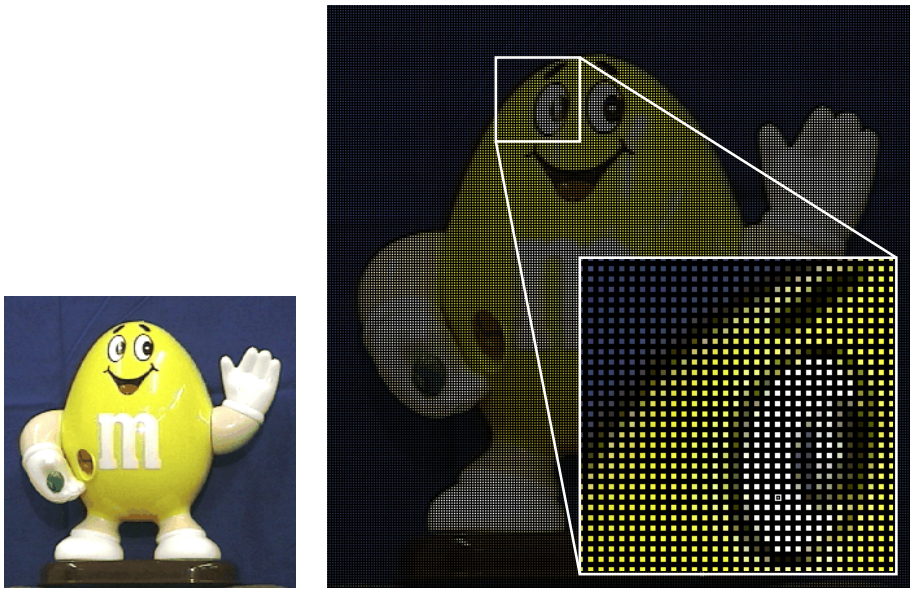


図 5.5: 画像の拡大・縮小

```
13    /* スケーリング */
14    for (row = TeoYstart (dst); row <= TeoYend (dst); row++) {
15      for (col = TeoXstart (dst); col <= TeoXend (dst); col++) {
16        in_row = (int) (row / yscale);
17        in_col = (int) (col / xscale);
18        for (p = 0; p < TeoPlane (dst); p++) {
19          TeoPutPixel (dst, col, row, 0, TEO_UINT8,
20                       TeoGetPixel (src, in_col, in_row, p, TEO_UINT8));
21        }
22      }
23    }
24    return dst;
25  }
```

5.4 例題 8 画素値の内挿

画像の拡大・縮小プログラムはできましたか? 拡大画像は解像度が低い荒いような感じの画像になっていませんか．それは画素座標が整数値であることが原因です．つまり画像を 2 倍に拡大する場合，出力画像の座標が $(1,0)$ である点の入力画像での座標は $(0.5,0)$ となります．しかし画素座標は整数値で表現されているため，座標を整数値に近似して，$(0,0)$ もしくは $(1,0)$ のどちらかの画素値を用いなければいけません．

これを解決する方法に双 1 次補間による画素値の内挿があります．計算によって得られた画素座標が図 5.6に示すように点 $(x,y),(x+1,y)$ を $\alpha:1-\alpha$ に，点 $(x,y),(x,y+1)$ を $\beta:1-\beta$ に内分する点である時，この点の画素値は式 (5.11) で計算することができます．

$$\begin{aligned} I(x+\alpha,y+\beta) &= (1-\beta)\Big((1-\alpha)I(x,y) + \alpha I(x+1,y)\Big) \\ &+ \beta\Big((1-\alpha)I(x,y+1) + \alpha I(x+1,y+1)\Big) \end{aligned} \tag{5.11}$$

変換によって得られた実数値の画素座標 $(\hat{x},\hat{y})$ から式 (5.11) 中の x,y,α,β は以下のように計算できます．ここで，関数 $floor$ は引数 $\hat{x}$ 以下の最大の整数値を返す関数です (ただし，戻り値の型は double)．この関数のプロトタイプ宣言は **math.h** に含まれています．関数 $floor$ の代わりに int 型へのキャストを使うことも可能です．

$$x = floor(\hat{x}) \tag{5.12}$$

$$y = floor(\hat{y}) \tag{5.13}$$

$$\alpha = \hat{x} - x \tag{5.14}$$

$$\beta = \hat{y} - y \tag{5.15}$$

リスト 5.4 に双一次補間による画素値の内挿の例を示します．この関数では，入力画像の画素値の型を TEO_UINT8 に限定しています．すべてのデータ型に対して同じ関数を用意するのは面倒です．これを解決

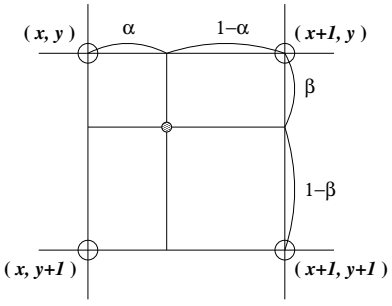


図 5.6: 画素値の内挿 (双 1 次補間)

する方法として，同じ内容をマクロとして定義する方法があります．マクロによる双一次補間をリスト 5.5 に示します．

リスト 5.4: 双一次補間による画素値の内挿

```
TEO_UINT8
TeoGetLinearPixelTEO_UINT8 (TEOIMAGE     *img,
                            double       x,
                            double       y,
                            int          plane) {
  TEO_UINT8     pix[4];
  int           ix, iy;
  double        a, b;

  ix = (int) x;
  iy = (int) y;
  a  = x - ix;
  b  = y - iy;

  pix[0] = TeoGetPixel (img, ix, iy, plane, TEO_UINT8);
  pix[1] = TeoGetPixel (img,
                        (ix >= TeoXend (img)) ?  ix : ix + 1,
                        iy,
                        plane, TEO_UINT8);
  pix[2] = TeoGetPixel (img,
                        ix,
                        (iy >= TeoYend (img)) ?  iy : iy + 1,
                        plane, TEO_UINT8);
  pix[3] = TeoGetPixel (img,
                        (ix >= TeoXend (img)) ?  ix : ix + 1,
                        (iy >= TeoYend (img)) ?  iy : iy + 1,
                        plane, TEO_UINT8);

  return (TEO_UINT8) ((1.0 - b) * (1.0 - a) * pix[0] +
                      (1.0 - b) * a         * pix[1] +
                      b         * (1.0 - a) * pix[2] +
                      b         * a         * pix[3]);
}
```

リスト 5.5: マクロによる双一次補間

```
#define TeoGetLinearPixel(image,index_x,index_y,index_p,ETYPE)\
  (((index_x)-((int) (index_x))) * \
   (((index_y)-((int) (index_y))) * \
    (double) TeoGetPixel(image, \
                         (((int) (index_x)) == (teo_xend(image))) ? \
                         ((int) (index_x)) : (((int) (index_x))+1), \
                         (((int) (index_y)) == (teo_yend(image))) ? \
                         ((int) (index_y)) : (((int) (index_y))+1), \
                         (index_p),ETYPE) + \
    (1.0-((index_y)-((int) (index_y)))) * \
    (double) TeoGetPixel(image, \
                         (((int) (index_x)) == (teo_xend(image))) ? \
                         ((int) (index_x)) : (((int) (index_x))+1), \
                         ((int) (index_y)), \
                         (index_p),ETYPE)) + \
   (1.0-((index_x)-((int) (index_x)))) * \
   (((index_y)-((int) (index_y))) * \
    (double) TeoGetPixel(image, \
                         ((int) (index_x)), \
                         (((int) (index_y)) == (teo_yend(image))) ? \
                         ((int) (index_y)) : (((int) (index_y))+1), \
                         (index_p),ETYPE) + \
    (1.0-((index_y)-((int) (index_y)))) * \
    (double) TeoGetPixel(image, \
                         ((int) (index_x)),((int) (index_y)), \
                         (index_p),ETYPE)))
```

(a) (b)

図 5.7: 画像の拡大 (a) 内挿なし (b) 内挿あり

5.5　例題９ パノラマ画像の生成

アルゴリズム編最後の課題はモザイク画像の生成です．モザイク画像とは複数の画像を張り合わせて生成した，より高視野な画像のことです．パノラマ画像という呼び方の方が一般的かもしれません．

平面または平面とみなせるもの (遠くのシーンを撮影したような場合) を異なる 2 視点から撮影したとき，この 2 画像間の変換は式 (5.16) で与えられます．この変換を**射影変換**と呼びます．

この変換は式 (5.17) のように書くことができ，このとき，式中の 3×3 行列を射影変換行列と呼びます．ここで，$Z[\cdot]$ はベクトルの第 3 成分を 1 とする (ベクトルの各成分を第 3 成分で割る) 正規化演算子を表します．この射影変換行列を利用して異なる視点から撮影した複数の画像を変換してつなぎ合わせることでモザイク画像を生成します．

$$\begin{cases} x' &= \dfrac{a_{11}x + a_{12}y + a_{13}}{a_{31}x + a_{32}y + a_{33}} \\ y' &= \dfrac{a_{21}x + a_{22}y + a_{23}}{a_{31}x + a_{32}y + a_{33}} \end{cases} \tag{5.16}$$

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = Z[\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}] \tag{5.17}$$

画像の拡大・縮小，画素値の内挿の課題で覚えたことを利用してモザイク画像を作成します．第 1 画像から第 2 画像への射影変換行列 $\boldsymbol{H}$ が与えられている場合を考えます．

1. 出力画像のサイズを決定します
   与えられた射影変換行列の逆変換 $\boldsymbol{H}^{-1}$ を求めて，第 2 画像の 4 隅の点が第 1 画像上のどの座標に変換されるか計算します．そして，第 1 画像の 4 隅の点と，変換された第 2 画像の 4 隅の点を内包する矩形領域を出力画像のサイズとします (図 5.8)．
   この際，出力画像の左上の座標が $(0, 0)$ でない場合も生じますが，TEO 画像フォーマットではオフセット機能により，そのような場合も問題なく扱うことができるので便利です．
2. 第 1 画像データを出力画像にコピーします
   第 1 画像と出力画像の座標系は一致していますので，第 1 画像上の画素値を出力画像の同一座標に書き込みます．
3. 第 2 画像データを出力画像にコピーします
   (a) 出力画像の各点について，射影変換行列 $\boldsymbol{H}$ により第 2 画像での座標を計算します．
   (b) このとき，計算された座標が第 2 画像の範囲外であれば何もしません．
   (c) 計算された座標が第 2 画像の範囲内であれば，双 1 次補間によって出力画像の画素値を計算し，書き込みます．

モザイク画像生成の例を図 5.9に示します．

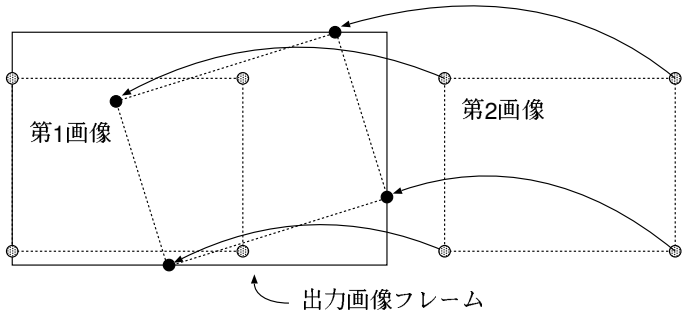


図 5.8: 画像サイズの決定

(a)

(b)

(c)

図 5.9: モザイク画像の生成結果 (a) 入力画像 1 (b) 入力画像 2 (c) モザイク画像

第 6 章

画像処理アルゴリズムレシピ

本章ではその他の画像処理アルゴリズムを紹介します.

6.1 画像の変換

ここでは，画像の平行移動や回転，拡大・縮小を含めた様々な画像の変換について説明します．これらの変換は全て行列として表現することが可能です.

6.1.1 画像変換の基本原理

画像を原点周りに角度 θ 回転する場合を考えます．このとき回転前の画素座標を (x, y)，回転後の画素座標を (x', y') とすると，次の式が成り立ちます (回転方向は座標形の取り方に依存します).

$$\left\{ \begin{array}{rcl} x' & = & x\cos\theta + y\sin\theta \\ y' & = & -x\sin\theta + y\cos\theta \end{array} \right. \tag{6.1}$$

しかし，前章の「拡大・縮小」でも扱ったように画像の変換は基本的には出力画像の画素座標 (x', y') から入力画像の画素座標 (x, y) を計算する逆変換によって行います.

従って画像の回転は次式を用いて行います.

$$\left\{ \begin{array}{rcl} x & = & x'\cos\theta - y'\sin\theta \\ y & = & x'\sin\theta + y'\cos\theta \end{array} \right. \tag{6.2}$$

変換後の画素座標は実数になることがありますので，その場合前章の「画素値の内挿」で説明した方法を用いて実数座標の画素の輝度値を計算します.

6.1.2 画像変換の一般形

さきほど説明した画像の回転は次のように行列で表現することができます.

$$\boldsymbol{x} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \boldsymbol{x}' \tag{6.3}$$

ここで $\boldsymbol{x}$, $\boldsymbol{x}'$ は画素座標 (x, y), (x', y') を次のように同次座標形でベクトル表現したものです.

$$\boldsymbol{x} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \quad \boldsymbol{x}' = \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} \tag{6.4}$$

上記のように画像の変換は基本的に 3×3 の行列で表現することが可能です.

6.1.3　いろいろな変換

以下に様々な画像変換の行列表現をまとめます.

- **並進** (t_x, t_y) の平行移動

$$\boldsymbol{x}' = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \boldsymbol{x}, \quad \boldsymbol{x} = \begin{pmatrix} 1 & 0 & -t_x \\ 0 & 1 & -t_y \\ 0 & 0 & 1 \end{pmatrix} \boldsymbol{x}' \tag{6.5}$$

- **回転** 原点周りの角度 θ の回転

$$\boldsymbol{x}' = \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \boldsymbol{x}, \quad \boldsymbol{x} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \boldsymbol{x}' \tag{6.6}$$

- **拡大・縮小** 原点を中心とする s 倍の拡大

$$\boldsymbol{x}' = \begin{pmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{pmatrix} \boldsymbol{x}, \quad \boldsymbol{x} = \begin{pmatrix} 1/s & 0 & 0 \\ 0 & 1/s & 0 \\ 0 & 0 & 1 \end{pmatrix} \boldsymbol{x}' \tag{6.7}$$

- **剛体運動** 原点周りに角度 θ だけ回転して，(t_x, t_y) の平行移動をする

$$\boldsymbol{x}' = \begin{pmatrix} \cos\theta & \sin\theta & t_x \\ -\sin\theta & \cos\theta & t_y \\ 0 & 0 & 1 \end{pmatrix} \boldsymbol{x}, \quad \boldsymbol{x} = \begin{pmatrix} \cos\theta & -\sin\theta & -t_x\cos\theta + t_y\sin\theta \\ \sin\theta & \cos\theta & -t_x\sin\theta - t_y\cos\theta \\ 0 & 0 & 1 \end{pmatrix} \boldsymbol{x}' \tag{6.8}$$

- **相似変換** 原点を中心に s 倍し，その周りに角度 θ だけ回転して，(t_x, t_y) の平行移動をする

$$\boldsymbol{x}' = \begin{pmatrix} s\cos\theta & s\sin\theta & t_x \\ -s\sin\theta & s\cos\theta & t_y \\ 0 & 0 & 1 \end{pmatrix} \boldsymbol{x}, \quad \boldsymbol{x} = \frac{1}{s} \begin{pmatrix} \cos\theta & -\sin\theta & -t_x\cos\theta + t_y\sin\theta \\ \sin\theta & \cos\theta & -t_x\sin\theta - t_y\cos\theta \\ 0 & 0 & s \end{pmatrix} \boldsymbol{x}' \tag{6.9}$$

- **アフィン変換**

$$\boldsymbol{x}' = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix} \boldsymbol{x}, \quad \boldsymbol{x} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix}^{-1} \boldsymbol{x}' \tag{6.10}$$

ただし，$\{a_{ij}\}$ は $a_{11}a_{22} - a_{21}a_{12} \neq 0$ となる任意の実数.

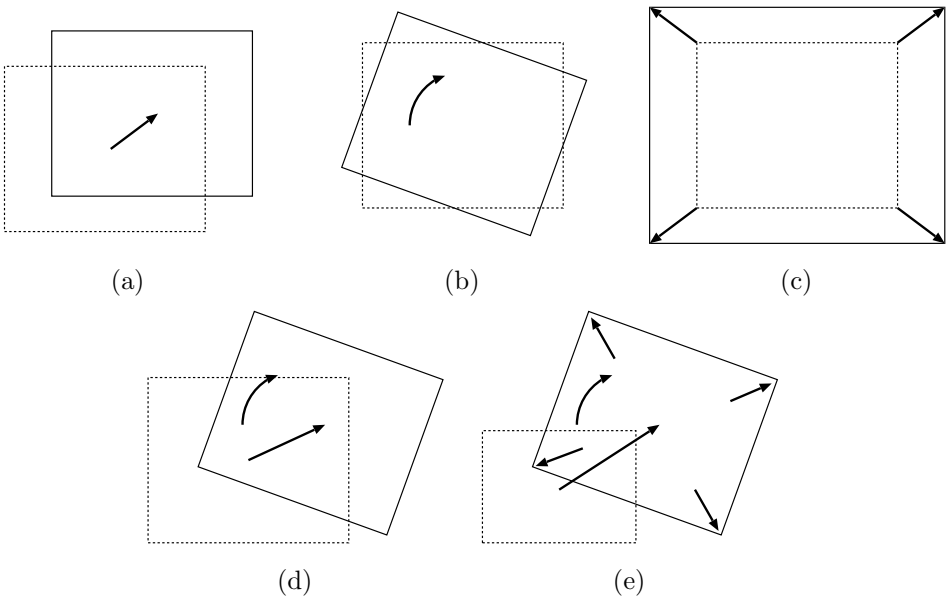


図 6.1: 画像の変換 (a) 並進 (b) 回転 (c) 拡大・縮小 (d) 剛体運動 (e) 相似変換

6.2　高速ガウシアンフィルタ

前章で画像を平滑化するガウシアンフィルタについて説明しましたが，これらのフィルタの計算量はそのフィルタ (マスク) サイズに依存して増加します．ここではガウシアンフィルタの計算を高速化するアルゴリズムを示します．

6.2.1　変数分離による処理の高速化

画像のフィルタ演算には，ガウシアンフィルタを含めてフィルタ $W(k,l)$ を次のように分離できるものがあります．

$$W(k,l) = W_1(k)W_2(l) \tag{6.11}$$

このようなフィルタを変数分離型 (または直積型) と呼びます．このとき，フィルタ演算は次のような 2 段階のフィルタ操作に分離することができます．

1. 入力画像 I に対して，次のようにして中間画像 $\hat{I}$ を作成する．

$$\hat{I}(x,y) = \sum_{-s \le k \le s} W_1(k)(x+k,y) \tag{6.12}$$

2. 中間画像 $\hat{I}$ に対して，次のようにして出力画像 I' を作成する．

$$I'(x,y) = \sum_{-s \le l \le s} W_2(l)(x,y+l) \tag{6.13}$$

6.2.2　変数分離型ガウシアンフィルタ

ガウシアンフィルタ $W(k,l) = e^{-(k^2+l^2)/2\sigma^2}$ を変数分離すると以下のようになります．

$$W(k,l) = e^{-(k^2+l^2)/2\sigma^2} = W_1(k)W_2(l) \tag{6.14}$$

$$W_1(k) = e^{-k^2/2\sigma^2}, \quad W_2(l) = e^{-l^2/2\sigma^2} \tag{6.15}$$

変数分離型ガウシアンフィルタのアルゴリズムは次のようになります．

1. フィルタ $W(k) = e^{-k^2/2\sigma^2}, -s \le k \le s$ の値を計算する．
2. 入力画像 I に対して，次のようにして中間画像 $\hat{I}$ を作成する．このときフィルタが画像からはみ出す部分の扱いに注意すること．

$$\hat{I}(x,y) = \sum_{-s \leq k \leq s} W(k)(x+k, y) \tag{6.16}$$

3. 中間画像 $\hat{I}$ に対して，次のようにして出力画像 I' を作成する．

$$I'(x,y) = \sum_{-s \leq l \leq s} W(l)(x, y+l) \tag{6.17}$$

6.3　エッジ検出

エッジとは画像中で輝度値が急激に変化する点の列であり，通常は物体の境界の象に対応します．ここでは，このようなエッジを画像中から検出するアルゴリズムを示します．

6.3.1　エッジ画像

画像 $I(x,y)$ を連続場とみなしたときの勾配 (グラジエント) は次のようになります．

$$\nabla I = \begin{pmatrix} I_x \\ I_y \end{pmatrix} \tag{6.18}$$

ただし，I_x, I_y は x, y に関する偏微分を表します．∇I の方向は輝度値が最も急激に変化する方向であり，そのノルム $||\nabla I||$ がその方向の輝度値の変化率を表します．エッジ画像とは各点で

$$||\nabla I|| = \sqrt{I_x^2 + I_y^2} \tag{6.19}$$

を輝度値とする画像とします．これをしきい値処理し，$||\nabla I||$ があるしきい値より大きい領域を取り出したものがエッジ領域となります．

6.3.2　平滑微分フィルタ

エッジを検出するためには輝度値を微分する必要があります．しかしこれを直接差分に置き換えると不規則な輝度値の変動の影響により，望ましくないエッジが多く得られることになります．これを避けるために，まず画像を平滑化して，その後に差分を行います．

これは次のような 1 つのフィルタ操作により行うことができます．

画像 $I(x,y)$ を連続場とみなしたときの重み関数 $w(x,y)$ による平滑化を x で微分したものは，分母を無視すると次のように表すことができます．

$$\begin{aligned} I'_x(x,y) &\propto -\int_{\infty}^{\infty}\int_{\infty}^{\infty} w_x(t-x, s-y) I(t,s) dt ds \\ &= -\int_{\infty}^{\infty}\int_{\infty}^{\infty} w_x(t', s') I(x+t', y+s') dt' ds' \end{aligned} \tag{6.20}$$

ただし，最後の項は $t' = t - x$, $s' = s - y$ と変数変換したものです．これを離散近似すると次のようになります．

$$I'_x(x,y) = -\frac{1}{C}\sum_{(k,l)\in\mathcal{N}} w_x(k,l)(x+k, y+l) \tag{6.21}$$

ここで C は定数であり，次のように定めています．

$$C = -\sum_{(k,l)\in\mathcal{N}} w_x(k,l)k \tag{6.22}$$

ガウス関数を変数 x, y でそれぞれ偏微分すると次のようになります．

$$w_x(x,y) = -\frac{x}{\sigma^2}e^{-(x^2+y^2)/2\sigma^2}, \quad w_y(x,y) = -\frac{y}{\sigma^2}e^{-(x^2+y^2)/2\sigma^2} \tag{6.23}$$

従って x 方向への平滑微分フィルタは次のようになります．

$$I'_x(x,y) = \sum_{(k,l)\in\mathcal{N}} D_x(k,l)I(x+k,y+l) \tag{6.24}$$

$$D_x(k,l) = \frac{ke^{-(k^2+l^2)/2\sigma^2}}{C}, \quad C = \sum_{(k,l)\in\mathcal{N}} k^2e^{-(k^2+l^2)/2\sigma^2} \tag{6.25}$$

同様に y 方向への平滑微分フィルタは次のようになります．

$$I'_y(x,y) = \sum_{(k,l)\in\mathcal{N}} D_y(k,l)I(x+k,y+l) \tag{6.26}$$

$$D_y(k,l) = \frac{le^{-(k^2+l^2)/2\sigma^2}}{C}, \quad C = \sum_{(k,l)\in\mathcal{N}} l^2e^{-(k^2+l^2)/2\sigma^2} \tag{6.27}$$

6.3.3 零交差法

連続した幅 1 画素の長いエッジを得るための方法として零交差法と呼ばれる方法があります．これは輝度値のラプラシアン

$$\Delta I = I_{xx} + I_{yy} \tag{6.28}$$

を各点の画素値とするラプラシアン画像を作成し，正の領域と負の領域の境界線を追跡するものです．

具体的にはラプラシアン画像 $\Delta I'$ の各画素 $\Delta I'(x,y)$ について，$\Delta I'(x,y)$, $\Delta I'(x+1,y)$, $\Delta I'(x,y+1)$, $\Delta I'(x+1,y+1)$ が同符号でなければ 1，そうでなければ 0 とする処理を行います．

6.3.4 平滑ラプラシアンフィルタ

平滑微分フィルタと同様に，画像を平滑化した後にラプラシアンフィルタをかけることで，不規則な輝度値の変動による影響を回避することができます．

画像 $I(x,y)$ を連続場とみなしたときの重み関数 $w(x,y)$ による平滑化結果にラプラシアンをかけたものは，次のように表すことができます．

$$\begin{aligned}\Delta I'(x,y) &\propto \int_{\infty}^{\infty}\int_{\infty}^{\infty} \Delta w(t-x, s-y) I(t,s) dt ds \\ &= \int_{\infty}^{\infty}\int_{\infty}^{\infty} \Delta w(t', s') I(x+t', y+s') dt' ds'\end{aligned} \tag{6.29}$$

ただし，最後の項は $t' = t - x$, $s' = s - y$ と変数変換したものです．これを離散近似すると次のようになります．

$$\Delta I'(x,y) = \frac{1}{C} \sum_{(k,l)\in\mathcal{N}} \Delta w(k,l)(x+k, y+l) \tag{6.30}$$

ここで C は定数であり，次のように定めています．

$$C = \frac{1}{4} \sum_{(k,l)\in\mathcal{N}} \Delta w(k,l)(k^2+l^2) \tag{6.31}$$

ガウス関数を 2 階微分して加えると，次のようになります．

$$\Delta w(x,y) = -\frac{2}{\sigma^2}\left(1 - \frac{x^2+y^2}{2\sigma^2}\right) e^{-(x^2+y^2)/2\sigma^2} \tag{6.32}$$

以上より平滑ラプラシアンフィルタは次のようになります．

$$\Delta I'(x,y) = \sum_{(k,l)\in\mathcal{N}} G(k,l) I(x+k, y+l) \tag{6.33}$$

$$G(k,l) = \frac{1}{C}\left(1 - \frac{x^2+y^2}{2\sigma^2}\right) e^{-(x^2+y^2)/2\sigma^2}, \quad C = \frac{1}{4}\sum_{(k,l)\in\mathcal{N}} (k^2+l^2)\left(1 - \frac{k^2+l^2}{2\sigma^2}\right) e^{-(k^2+l^2)/2\sigma^2} \tag{6.34}$$

6.4 テクスチャマッピング

画像から復元した 3 次元形状や CG により作成したワイヤフレームモデルに対して，各点を結んでできる多角形に画像を張り付けることをテクスチャマッピングと呼びます．

ここでは，画像中の指定した三角形領域を別の三角形領域にマッピングするアルゴリズムを示します．

6.4.1 重心座標

3 点 $A:(x_A,y_A)$, $B:(x_B,y_B)$, $C:(x_C,y_C)$ に関する**重心座標**とは，点 (x,y) を次のように表すものです．

$$\begin{pmatrix} x \\ y \end{pmatrix} = \alpha \begin{pmatrix} x_A \\ y_A \end{pmatrix} + \beta \begin{pmatrix} x_B \\ y_B \end{pmatrix} + \gamma \begin{pmatrix} x_C \\ y_C \end{pmatrix}, \quad \alpha+\beta+\gamma = 1 \tag{6.35}$$

式 (6.35) は次のように書き直すことができます．

$$\begin{pmatrix} x_A & x_B & x_C \\ y_A & y_B & y_C \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{6.36}$$

このとき，点 (x,y) に対する (α,β,γ) は次のように求めることができます．

$$\alpha = \frac{1}{D} \begin{vmatrix} x & x_B & x_C \\ y & y_B & y_C \\ 1 & 1 & 1 \end{vmatrix} = \frac{(y_B-y_C)x - (x_B-x_C)y + (x_By_C - y_Bx_C)}{D} \tag{6.37}$$

$$\beta = \frac{1}{D} \begin{vmatrix} x_A & x & x_C \\ y_A & y & y_C \\ 1 & 1 & 1 \end{vmatrix} = \frac{(y_C-y_A)x - (x_C-x_A)y + (x_Cy_A - y_Cx_A)}{D} \tag{6.38}$$

$$\gamma = \frac{1}{D} \begin{vmatrix} x_A & x_B & x \\ y_B & y_B & y \\ 1 & 1 & 1 \end{vmatrix} = \frac{(y_A-y_B)x - (x_A-x_B)y + (x_Ay_B - y_Ax_B)}{D} \tag{6.39}$$

$$D = \begin{vmatrix} x_A & x_B & x_C \\ y_B & y_B & y_C \\ 1 & 1 & 1 \end{vmatrix} = (x_By_C - y_Bx_C) + (x_Cy_A - y_Cx_A) + (x_Ay_B - y_Ax_B) \tag{6.40}$$

6.4.2 三角形領域のマッピング

三角形 ABC 内の画像を三角形 $A'B'C'$ 内に描画するアルゴリズムを示します．

1. 三角形 $A'B'C'$ に外接する矩形領域を求める．

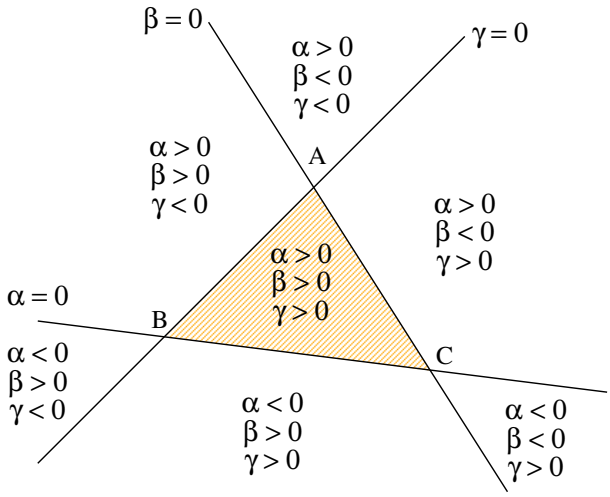


図 6.2: 重心座標の符号

$$x_{min} \leq x \leq x_{max}, \quad y_{min} \leq y \leq y_{max} \tag{6.41}$$

2. 次の値を計算する．

$$\begin{aligned} a_{11} &= y_B - y_C, \quad a_{12} = x_B - x_C, \quad a_{13} = x_B y_C - y_B x_C \\ a_{21} &= y_C - y_A, \quad a_{22} = x_C - x_A, \quad a_{23} = x_C y_A - y_C x_A \end{aligned} \tag{6.42}$$

$$D = a_{11} + a_{23} + x_A y_B - y_A x_B \tag{6.43}$$

3. 矩形領域 $(x_{min}, y_{min}) - (x_{max}, y_{max})$ 内の全ての点 (x, y) に対して，以下の処理を行う．

（a）点 (x, y) の重心座標 (α, β, γ) を次の式によって計算する．

$$\alpha = \frac{a_{11}x - a_{12}y + a_{13}}{D}, \quad \beta = \frac{a_{21}x - a_{22}y + a_{23}}{D}, \quad \gamma = 1 - \alpha - \beta \tag{6.44}$$

（b）$\alpha \geq 0, \beta \geq 0, \gamma \geq 0$ であれば三角形 ABC 内の点 (x', y') を式 (6.35) で計算する．

（c）点 (x', y') の画素値を点 (x, y) にコピーする．

6.5　2 値化処理 - 大津の方法による自動しきい値計算

濃淡画像を 2 値化するしきい値を自動的に求める方法として，判別分析による方法 (以下，大津の方法と呼ぶ) が有名です．大津の方法は，濃淡画像の輝度値ヒストグラムが地の部分と図の部分でそれぞれ山を作る (双峰性) と仮定し，地と図の集まり (以下，クラスと呼ぶ) の分離度が一番高くなるしきい値を計算する方法です．

大津の方法では，各輝度値をしきい値としたときの各クラス内の分散 σ_W^2 とクラス間の分散 σ_B^2 を計算し，クラス内での分散とクラス間の分散の比 σ_W^2/σ_B^2 が最小となる輝度値をしきい値としています．

6.5.1　輝度値ヒストグラムの作成

対象の画像は，画像サイズ $M \times N$，L 階調であるとし，以下の手順で輝度値ヒストグラムを作成します．

1. 長さ L の配列 $w(l), l = 0, \ldots, L-1$ を用意し，全要素を 0 に初期化する．
2. 入力画像の各画素を順に走査し，その輝度値が l であれば $w(l) \leftarrow w(l) + 1$ と 1 を加える．
3. $l = 0$ から $w(l)$ の値を順に調べ，$w(l) > 0$ となる最小の l を求め，l_{min} とする．
4. $l = L - 1$ から $w(l)$ の値を順に調べ，$w(l) > 0$ となる最大の l を求め，l_{max} とする．

6.5.2　しきい値の探索

まず各変数を次のように初期化する．

$$l = l_{min} - 1 \tag{6.45}$$

$$n_1 = 0,\ n_2 = MN \tag{6.46}$$

$$s_1 = 0.0,\ s_2 = \sum_{i=l_{min}}^{l_{max}} i \times w(i) \tag{6.47}$$

$$t_1 = 0.0,\ t_2 = \sum_{i=l_{min}}^{l_{max}} i \times i \times w(i) \tag{6.48}$$

$$\lambda' = 0.0 \tag{6.49}$$

以下の操作を繰り返し，しきい値を求める．

1. $l = l_{max}$ であれば l を返して終了する．
2. $w(l) = 0$ であれば $l \leftarrow l + 1$ としてステップ 1 に戻る．
3. 次のように変数を更新する．

$$l \leftarrow l + 1 \tag{6.50}$$

$$z = w(l),\ z' = w(l) \times l,\ z'' = w(l) \times l \times l \tag{6.51}$$

$$n_1 \leftarrow n_1 + z,\ s_1 \leftarrow s_1 + z',\ t_1 \leftarrow t_1 + z'' \tag{6.52}$$

$$n_2 \leftarrow n_2 - z,\ s_2 \leftarrow s_2 - z',\ t_2 \leftarrow t_2 - z'' \tag{6.53}$$

4. 2 クラスの平均 μ_1, μ_2 と分散 σ_1, σ_2 を計算する．

$$\mu_1 = \frac{s_1}{n_1},\ \sigma_1^2 = \frac{t_1}{n_1} - \mu_1^2,\ \mu_2 = \frac{s_2}{n_2},\ \sigma_2^2 = \frac{t_2}{n_2} - \mu_2^2 \tag{6.54}$$

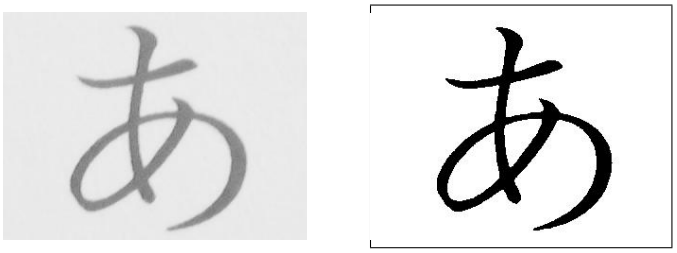
5. クラス間分散 σ_B^2 とクラス内分散 σ_W^2 を計算する．

$$\sigma_B^2 = n_1 n_2 (\mu_1 - \mu_2)^2,\ \sigma_W^2 = n_1 \sigma_1^2 + n_2 \sigma_2^2 \tag{6.55}$$

6. クラス間分散とクラス内分散の比を計算する．

$$\lambda = \frac{\sigma_B^2}{\sigma_W^2} \tag{6.56}$$

7. $\lambda < \lambda'$ なら l を返して終了する．そうでなければ $\lambda' \leftarrow \lambda$ としてステップ 1 に戻る．



(a) 入力濃淡画像　　(b) 大津の方法による 2 値化画像

図 6.3: 大津の方法によって計算したしきい値による 2 値化結果

付録 A

libteo 関数リファレンス

A.1 ファイルアクセス関数

```
TEOFILE* TeoOpenFile (char *filename);
```

引数

filename： TEO 画像ファイル名

説明

引数で与えられた TEO 画像ファイルをオープンします.
ファイル名が”-” の場合，標準入力から読み込みます．ファイルが PNM フォーマットの場合は自動的に判定し TEO ファイルと同様に扱うことができます.

戻り値

画像情報を書き込んだ TEOFILE 構造体へのポインタ.
画像のオープンに失敗した場合は NULL を返します.

```
int TeoCloseFile (TEOFILE *teofp);
```

引数

teofp： TEOFILE 構造体へのポインタ

説明

引数で与えられた TEOFILE をクローズします.

戻り値

TEOFILE のクローズが成功した場合は 1，失敗した場合には 0 を返します.

```
TEOFILE* TeoCreateFile (char *filename,
                        int  width,
                        int  height,
                        int  xoffset,
                        int  yoffset,
                        int  type,
                        int  bit,
                        int  plane,
                        int  frame);
```

引数

filename ：画像ファイル名
width ：画像の幅
height ：画像の高さ
xoffset ：オフセットの X 座標
yoffset ：オフセットの Y 座標
type ：画素値の型
bit ：画素値のビット数
plane ：プレーン数
frame ：フレーム数

説明

指定されたパラメータにより新規に TEO ファイルを作成します．
ファイル名が”-” の場合，標準出力に書き出されます．

戻り値

作成した TEOFILE 構造体へのポインタ．TEOFILE 構造体の作成に失敗した場合は NULL を返します．

参照

TeoCreateFileWithUserExtension

```
TEOFILE* TeoCreateSimilarFile (char    *filename,
                               TEOFILE *teofp);
```

引数

filename ：画像ファイル名
teofp ：TEOFILE へのポインタ

説明

引数に与えた TEOFILE 構造体 teofp と同じパラメータで新規に TEO ファイルを作成します．
ファイル名が”-” の場合，標準出力に書き出されます．

戻り値

作成した TEOFILE 構造体へのポインタ．TEOFILE 構造体の作成に失敗した場合は NULL を返します．

```
TEOFILE* TeoCreateFileWithUserExtension (char *filename,
                                         int  width,
                                         int  height,
                                         int  xoffset,
                                         int  yoffset,
                                         int  type,
                                         int  bit,
                                         int  plane,
                                         int  frame,
                                         int  extc,
                                         char **extv);
```

引数

filename	: 画像ファイル名
width	: 画像の幅
height	: 画像の高さ
xoffset	: オフセットの X 座標
yoffset	: オフセットの Y 座標
type	: 画素値の型
bit	: 画素値のビット数
plane	: プレーン数
frame	: フレーム数
extc	: ユーザ拡張項目の数
extv	: ユーザ拡張項目へのポインタ

説明

指定されたパラメータにより新規にユーザ拡張付きの TEO ファイルを作成します.
ファイル名が"-" の場合，標準出力に書き出されます.
extc はユーザ拡張項目の数を指定し，extv には予め extc 個のユーザ拡張項目を登録しておく必要があります.

戻り値

作成した TEOFILE 構造体へのポインタ．TEOFILE 構造体の作成に失敗した場合は NULL を返します.

参照

TeoCreateFile
TeoGetUserExtension

A.2 画像アクセス関数

```
TEOIMAGE* TeoAllocImage (int  width,
                         int  height,
                         int  xoffset,
                         int  yoffset,
                         int  type,
                         int  bit,
                         int  plane);
```

引数

width ：画像の幅
height ：画像の高さ
xoffset ：オフセットの X 座標
yoffset ：オフセットの Y 座標
type ：画素値の型
bit ：画素値のビット数
plane ：プレーン数

説明

指定されたパラメータにより画像データ用のメモリ領域を確保します.

戻り値

確保したメモリ領域へのポインタ．メモリ確保に失敗した場合は NULL を返します.

```
TEOIMAGE* TeoAllocSimilarImage (TEOFILE *teofp);
```

引数

teofp ：TEOFILE 構造体へのポインタ

説明

引数で与えた TEOFILE 構造体 teofp と同じパラメータで画像データ用のメモリ領域を確保します.

戻り値

確保したメモリ領域へのポインタ．メモリ確保に失敗した場合は NULL を返します.

```
TEOIMAGE* TeoAllocSimilarImage (TEOIMAGE *teoimg);
```

引数

teoimg ：TEOIMAGE 構造体へのポインタ

説明

引数で与えた TEOIMAGE 構造体 teoimg と同じパラメータで画像データ用のメモリ領域を確保します.

戻り値

確保したメモリ領域へのポインタ．メモリ確保に失敗した場合は NULL を返します.

```
int TeoFreeImage (TEOIMAGE *teoimg);
```

引数

teoimg ：TEOIMAGE 構造体へのポインタ

説明

引数で与えた TEOIMAGE 構造体のメモリ領域を解放します.

戻り値

成功した場合は 1，失敗した場合には 0 を返します.

```
int TeoReadFrame (TEOFILE  *teofp,
                  TEOIMAGE *teoimg);
```

引数

teofp ：TEOFILE へのポインタ
teoimg ：TEOIMAGE へのポインタ

説明

現在のファイルポインタの位置から，1 フレーム分の画像データを読み込み，TEOIMAGE 構造体のデータ領域にコピーします.
この関数を実行した後は，ファイルポインタは次のフレームの先頭に移動します.

戻り値

成功した場合は 1，失敗した場合には 0 を返します.

```
int TeoWriteFrame (TEOFILE  *teofp,
                   TEOIMAGE *teoimg);
```

引数

teofp ：TEOFILE へのポインタ
teoimg ：TEOIMAGE へのポインタ

説明

現在のファイルポインタの位置に，指定されたメモリ領域にある 1 フレーム分の画像データを書き出します.
この関数を実行した後は，ファイルポインタは次のフレームの先頭に移動します.

戻り値

成功した場合は 1，失敗した場合には 0 を返します.

```
int TeoSetAbsFrame (TEOFILE *teofp,
                    int     frame);
```

引数

teofp : TEOFILE へのポインタ
frame : フレーム番号

説明

引数で指定したフレーム番号の画像の先頭にファイルポインタを移動します.
0 を指定した場合は最初のフレームの先頭アドレスにファイルポインタが移動します.

戻り値

成功した場合は 1, 失敗した場合には 0 を返します.

参照

TeoSetRelFrame

```
int TeoSetRelFrame (TEOFILE *teofp,
                    int     frame);
```

引数

teofp : TEOFILE へのポインタ
frame : 移動フレーム数

説明

引数で指定したフレーム数だけ相対的にファイルポインタを移動します.
0 を指定した場合はファイルポインタは移動しません.

戻り値

成功した場合は 1, 失敗した場合には 0 を返します.

参照

TeoSetAbsFrame

```
int TeoCheckFrame (TEOFILE *teofp);
```

引数

teofp : TEOFILE へのポインタ

説明

次にアクセスしようとするフレームが引数に与えた画像のフレーム数内にあるかどうか調べます.

返り値

次にアクセスしようとするフレームが引数に与えた画像のフレーム数内であれば真 (非 0 値),
そうでなければ偽 (0) を返します.

A.3 画素アクセス関数

```
ETYPE TeoGetPixel (TEOIMAGE *teoimg,
                   int      x,
                   int      y,
                   int      plane,
                   ETYPE);
```

引数

teoimg : TEOIMAGE へのポインタ
x : 画素の x 座標
y : 画素の y 座標
plane : プレーン番号
ETYPE : 画素値の型

説明

点 (x, y) の plane 番目のプレーンの画素値を取得します.
ETYPE には teoimg の画素値の型を指定します．また，第 1 プレーンのプレーン番号は 0 であることに注意して下さい.

戻り値

点 (x, y) の plane 番目のプレーンの画素値.

```
void TeoPutPixel (TEOIMAGE *teoimg,
                  int      x,
                  int      y,
                  int      plane,
                  ETYPE,
                  ETYPE    val);
```

引数

teoimg : TEOIMAGE へのポインタ
x : 画素の x 座標
y : 画素の y 座標
plane : プレーン番号
ETYPE : 画素値の型
val : 画素値

説明

点 (x, y) の plane 番目のプレーンに指定した画素値 val を書き込みます.
ETYPE には teoimg の画素値の型を指定します．また，第 1 プレーンのプレーン番号は 0 であることに注意して下さい.

戻り値

なし (void)

```
TEO_BIT TeoGetBit (TEOIMAGE *teoimg,
                   int      x,
                   int      y,
                   int      plane);
```

引数

teoimg : TEOIMAGE へのポインタ
x : 画素の x 座標
y : 画素の y 座標
plane : プレーン番号

説明

2 値 (TEO_BIT 型) 画像の点 (x, y)，plane 番目のプレーンの画素値を取得します.
第 1 プレーンのプレーン番号は 0 であることに注意して下さい.

戻り値

点 (x, y) の plane 番目のプレーンの画素値 (0 or 1).

```
void TeoPutBit (TEOIMAGE *teoimg,
                int      x,
                int      y,
                int      plane,
                TEO_BIT val);
```

引数

teoimg : TEOIMAGE へのポインタ
x : 画素の x 座標
y : 画素の y 座標
plane : プレーン番号
val : 画素値

説明

2 値 (TEO_BIT 型) 画像の点 (x, y)，plane 番目のプレーンに指定した画素値 val を書き込みます.
また，第 1 プレーンのプレーン番号は 0 であることに注意して下さい.

戻り値

なし (void)

A.4 画像情報アクセス関数

```
int TeoWidth (TEOFILE  *teofp);
int TeoWidth (TEOIMAGE *teoimg);
```

引数

teofp : TEOFILE へのポインタ
teoimg : TEOIMAGE へのポインタ

説明 & 戻り値

画像の幅を返します.

```
int TeoHeight (TEOFILE  *teofp);
int TeoHeight (TEOIMAGE *teoimg);
```

引数

teofp : TEOFILE へのポインタ
teoimg : TEOIMAGE へのポインタ

説明 & 戻り値

画像の高さを返します.

```
int TeoType (TEOFILE  *teofp);
int TeoType (TEOIMAGE *teoimg);
```

引数

teofp : TEOFILE へのポインタ
teoimg : TEOIMAGE へのポインタ

説明 & 戻り値

画素値の型 (**TEO_SIGNED**, **TEO_UNSIGNED**, **TEO_FLOAT**) を返します.

```
int TeoBit (TEOFILE  *teofp);
int TeoBit (TEOIMAGE *teoimg);
```

引数

teofp : TEOFILE へのポインタ
teoimg : TEOIMAGE へのポインタ

説明 & 戻り値

画素値のビット数を返します.

```
int TeoPlane (TEOFILE  *teofp);
int TeoPlane (TEOIMAGE *teoimg);
```

引数

teofp : TEOFILE へのポインタ
teoimg : TEOIMAGE へのポインタ

説明 & 戻り値

画像のプレーン数を返します.

```
int TeoFsize (TEOFILE  *teofp);
int TeoFsize (TEOIMAGE *teoimg);
```

引数

teofp : TEOFILE へのポインタ
teoimg : TEOIMAGE へのポインタ

説明 & 戻り値

画像データサイズ (幅 × 高さ × プレーン数) を返します.

```
int TeoXoffset (TEOFILE  *teofp);
int TeoXoffset (TEOIMAGE *teoimg);
```

引数

teofp : TEOFILE へのポインタ
teoimg : TEOIMAGE へのポインタ

説明 & 戻り値

オフセットの X 座標を返します.

```
int TeoYoffset (TEOFILE  *teofp);
int TeoYoffset (TEOIMAGE *teoimg);
```

引数

teofp : TEOFILE へのポインタ
teoimg : TEOIMAGE へのポインタ

説明 & 戻り値

オフセットの Y 座標を返します.

```
int TeoXstart (TEOFILE  *teofp);
int TeoXstart (TEOIMAGE *teoimg);
```

引数

teofp : TEOFILE へのポインタ
teoimg : TEOIMAGE へのポインタ

説明 & 戻り値

画像の左上点の X 座標を返します.

```
int TeoYstart (TEOFILE  *teofp);
int TeoYstart (TEOIMAGE *teoimg);
```

引数

teofp : TEOFILE へのポインタ
teoimg : TEOIMAGE へのポインタ

説明 & 戻り値

画像の左上点の Y 座標を返します.

```
int TeoXend (TEOFILE  *teofp);
int TeoXend (TEOIMAGE *teoimg);
```

引数

teofp : TEOFILE へのポインタ
teoimg : TEOIMAGE へのポインタ

説明 & 戻り値

画像の右下点の X 座標を返します.

```
int TeoYend (TEOFILE  *teofp);
int TeoYend (TEOIMAGE *teoimg);
```

引数

teofp : TEOFILE へのポインタ
teoimg : TEOIMAGE へのポインタ

説明 & 戻り値

画像の右下点の Y 座標を返します.

```
int TeoIsBIT (TEOFILE  *teofp);
int TeoIsBIT (TEOIMAGE *teoimg);
```

引数

teofp : TEOFILE へのポインタ
teoimg : TEOIMAGE へのポインタ

説明 & 戻り値

画素値の型が **TEO_BIT** であれば真 (非 0 値)，そうでなければ偽 (0) を返します.

```
int TeoIsUINT8 (TEOFILE  *teofp);
int TeoIsUINT8 (TEOIMAGE *teoimg);
```

引数

teofp : TEOFILE へのポインタ
teoimg : TEOIMAGE へのポインタ

説明 & 戻り値

画素値の型が **TEO_UINT8** であれば真 (非 0 値)，そうでなければ偽 (0) を返します.

```
int TeoIsSINT8 (TEOFILE  *teofp);
int TeoIsSINT8 (TEOIMAGE *teoimg);
```

引数

teofp : TEOFILE へのポインタ
teoimg : TEOIMAGE へのポインタ

説明 & 戻り値

画素値の型が **TEO_SINT8** であれば真 (非 0 値)，そうでなければ偽 (0) を返します.

```
int TeoIsUINT16 (TEOFILE  *teofp);
int TeoIsUINT16 (TEOIMAGE *teoimg);
```

引数

teofp : TEOFILE へのポインタ
teoimg : TEOIMAGE へのポインタ

説明 & 戻り値

画素値の型が **TEO_UINT16** であれば真 (非 0 値)，そうでなければ偽 (0) を返します.

```
int TeoIsSINT16 (TEOFILE  *teofp);
int TeoIsSINT16 (TEOIMAGE *teoimg);
```

引数

teofp : TEOFILE へのポインタ
teoimg : TEOIMAGE へのポインタ

説明 & 戻り値

画素値の型が **TEO_SINT16** であれば真 (非 0 値)，そうでなければ偽 (0) を返します.

```
int TeoIsUINT32 (TEOFILE  *teofp);
int TeoIsUINT32 (TEOIMAGE *teoimg);
```

引数

teofp : TEOFILE へのポインタ
teoimg : TEOIMAGE へのポインタ

説明 & 戻り値

画素値の型が **TEO_UINT32** であれば真 (非 0 値)，そうでなければ偽 (0) を返します.

```
int TeoIsSINT32 (TEOFILE  *teofp);
int TeoIsSINT32 (TEOIMAGE *teoimg);
```

引数

teofp : TEOFILE へのポインタ
teoimg : TEOIMAGE へのポインタ

説明 & 戻り値

画素値の型が **TEO_SINT32** であれば真 (非 0 値)，そうでなければ偽 (0) を返します.

```
int TeoIsFLOAT32 (TEOFILE  *teofp);
int TeoIsFLOAT32 (TEOIMAGE *teoimg);
```

引数

teofp : TEOFILE へのポインタ
teoimg : TEOIMAGE へのポインタ

説明 & 戻り値

画素値の型が **TEO_FLOAT32** であれば真 (非 0 値)，そうでなければ偽 (0) を返します.

```
int TeoIsFLOAT64 (TEOFILE  *teofp);
int TeoIsFLOAT64 (TEOIMAGE *teoimg);
```

引数

teofp : TEOFILE へのポインタ
teoimg : TEOIMAGE へのポインタ

説明 & 戻り値

画素値の型が **TEO_FLOAT64** であれば真 (非 0 値)，そうでなければ偽 (0) を返します.

```
int TeoFrame (TEOFILE  *teofp);
```

引数

teofp : TEOFILE へのポインタ

説明 & 戻り値

フレーム数を返します.

```
int TeoHsize (TEOFILE  *teofp);
```

引数

teofp : TEOFILE へのポインタ

説明 & 戻り値

データ部の先頭ポイントを返します.

```
FILE* TeoFp (TEOFILE  *teofp);
```

引数

teofp : TEOFILE へのポインタ

説明 & 戻り値

TEO ファイルへのファイルポインタを返します.

```
int TeoExtc (TEOFILE  *teofp);
```

引数

teofp : TEOFILE へのポインタ

説明 & 戻り値

ユーザ拡張の項目数を返します.

```
char** TeoExtv (TEOFILE  *teofp);
```

引数

teofp ：TEOFILE へのポインタ

説明 & 戻り値

ユーザ拡張の拡張項目へのポインタを返します.

```
char* TeoGetUserExtension (TEOFILE *teofp,
                           char    *key);
```

引数

teofp ：TEOFILE へのポインタ

key ：ユーザ拡張キー

説明

引数に与えられたユーザ拡張キーと一致する項目のユーザ拡張アイテムを検索します.

戻り値

引数に与えられたユーザ拡張キーと一致する項目のユーザ拡張アイテムを返します.

```
int TeoCurrent (TEOFILE  *teofp);
```

引数

teofp ：TEOFILE へのポインタ

説明 & 戻り値

次にアクセスするフレーム番号を返します.

```
void* TeoData (TEOIMAGE *teoimg);
```

引数

teoimg ：TEOIMAGE へのポインタ

説明 & 戻り値

画像データへのポインタを返します.

付録 B

便利な TEO コマンド

B.1 pnm2teo

pnm2teo *input.pnm*

引数

input.pnm ：入力 PNM 画像ファイル名

説明

入力 PNM 画像を TEO 画像に変換し，結果を標準出力に出力します.

使用例

% pnm2teo input.pnm > output.teo

B.2 teo2pnm

teo2pnm *#frame_number input.teo*

引数

frame_number ：変換する画像データのフレーム番号

input.teo ：入力 TEO 画像ファイル名

説明

入力 TEO 画像の指定したフレームデータを PNM 画像に変換し，結果を標準出力に出力します.

使用例

% teo2pnm 0 input.teo > output.ppm

B.3　teo2avi

teo2avi　*input.teo output.avi*

引数

input.teo　：入力 TEO 画像ファイル名
output.avi　：出力 AVI ファイル名

オプション

-c codec　：使用するコーデック
-q quality　：画質 (0–100)
-s start end step　：複数ファイルを入力とする場合に使用する．
-r frame_rate　：フレームレート (標準 30[fps])

説明

入力 TEO 画像シーケンスを AVI フォーマットに変換します．

使用例

% teo2avi input.teo output.avi
% teo2avi input%02d.teo output.avi -s 0 10 1

B.4　teogzip

teogzip [-d]　*input.teo*

引数

input.teo　：入力 TEO 画像ファイル名

オプション

-d　：ファイルの伸長を行うオプション

説明

TEO 画像の圧縮を行います．-d オプションを付けると圧縮された入力 TEO 画像を伸長します．
圧縮・伸長後のファイル名は入力ファイル名と同一になります．

使用例

% teogzip uncompressed.teo
% teogzip -d compressed.teo

B.5 teo_get_extension

teo_get_extension *input.teo* [keyword]

引数

input.teo ：入力 TEO 画像ファイル名

オプション

keyword ：ユーザ拡張キー

説明

引数に与えた TEO 画像ファイルのヘッダ部を読んで，ユーザ拡張項目があればそれを表示します．
オプションにユーザ拡張キーを指定した場合，ユーザ拡張キーと一致するユーザ拡張項目を探し，値を表示します．

使用例

% teo_get_extension has_extension.teo
HasAlpha YES
% teo_get_extension has_extension.teo HasAlpha
YES

B.6 teocast

teocast *pixel-type bits* [input.teo]

引数

pixel-type ：画素値の型 (U, S, F のいづれかを指定)
bits ：画素値のビット数 (1, 8, 16, 32, 64 のいづれかを指定)

オプション

input.teo ：入力画像ファイル名

説明

入力画像を引数に与えた画素値の型へ変換します．入力画像ファイル名を省略した場合は標準入力からデータを読み込みます．

使用例

% teocast U 8 FLOAT64.teo > UINT8.teo
% cat FLOAT64.teo | teocast U 8 > UINT8.teo

B.7 teodiff

teodiff `[-s|-l1|-l2|-c] [-abs]` *input1.teo input2.teo*

引数

input1.teo : 入力 TEO 画像ファイル 1
input2.teo : 入力 TEO 画像ファイル 2

オプション

-s : マルチプレーンの場合，画素ごとのベクトル差分を求めます．
-l1 : マルチプレーンの場合，画素ごとの L1 ノルムを求めます．
-l2 : マルチプレーンの場合，画素ごとの L2 ノルムを求めます．
-c : マルチプレーンの場合，画素ごとのベクトル間の余弦を求めます．
-abs : 結果が負の値の場合，絶対値を求めます．

説明

引数に与えた 2 画像の差分画像を作成し，結果を標準出力に出力します．出力画像の画素値の型は TEO_FLOAT64 になります．

使用例

% teodiff input.teo background.teo > diff.teo

B.8 teorange

teorange *srcmin srcmax destmin destmax* `[input.teo]`

引数

srcmin : 説明を参照して下さい．
srcmax : 説明を参照して下さい．
destmin : 説明を参照して下さい．
destmax : 説明を参照して下さい．

オプション

input.teo : 入力 TEO 画像ファイル名

説明

入力画像の画素値の範囲を [srcmin:srcmax] から [destmin:destmax] に線形に変換し，標準出力に出力します．出力画像の画素値の型は TEO_FLOAT64 になります．

使用例

% teorange 0 100 0 255 input.teo > output.teo

付録 C

TEO 画像ビューワ TeoEyes

TeoEyes は TEO 画像用に開発された画像ビューワです．TEO 画像フォーマット以外の画像フォーマットにも対応し，画像表示だけでなく簡単なフィルタリング等も行えます．もちろんマルチフレームの TEO 画像にも対応していて，TEO 動画像の再生も可能です．

TeoEyes を画像ファイル名を指定せずに起動すると図 C.1のような画像が表示されます．ユーザの設定で起動時に表示される画像を変更することも可能です．

また TeoEyes では図 C.2のようにディレクトリ内にある画像を一覧表示することもできます．また，輝度補正やサイズ変更，画像の切りだし，画面のキャプチャなどを行うことも可能です (図 C.3)．

図 C.1: TeoEyes の起動画面

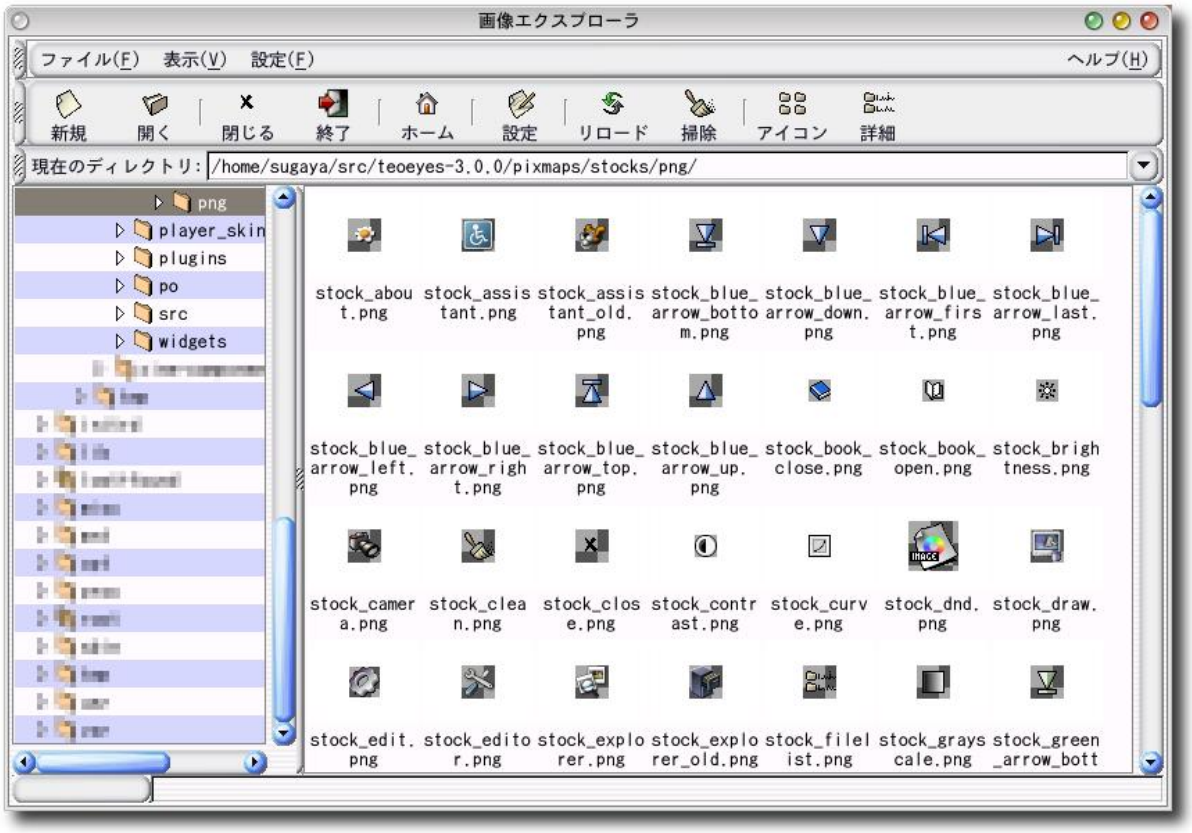


図 C.2: 画像の一覧表示

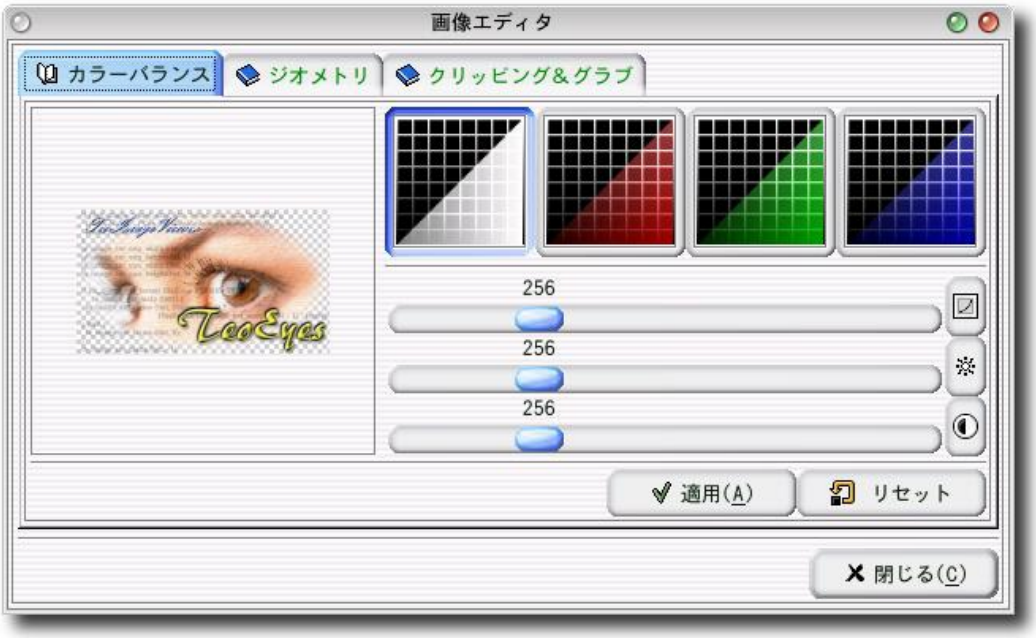


図 C.3: 画像の編集画面

付録 D

TEO 拡張ライブラリ

TEO に関するライブラリには基本ライブラリである libteo をはじめ，それ以外にも様々なライブラリが存在します．ここでは，その中の代表的なライブラリを紹介します．

D.1 libteopp

libteopp は libteo で定義された TEOFILE や TEOIMAGE をクラス化した C++ 版の TEO ライブラリです．

D.2 ruby-teo

ruby-teo は libteo の ruby 版です．スクリプト言語 Ruby を使用するため，プログラムをコンパイルすることなく，手軽に扱うことが可能です．

D.3 libteo2ipl

libteo2ipl は TEO 画像データを OpenCV で扱うデータ形式に変換するライブラリです．

OpenCV には様々な画像処理アルゴリズムが実装されており，OpenCV を利用することにより，これらのアルゴリズムをユーザが独自に実装することなく使用することができます．

D.4 libteo+

libteo+ は libteo には含まれない画像処理によく使用される関数を集めた拡張ライブラリです．libteo+ には次のような関数が含まれます．

- **画素アクセスの拡張関数**
  双 1 次補間によりサブピクセルの画素値を求める関数等
- **レンジチェック関数**
  指定した座標が画像の範囲内にあるかどうか調べる関数や画像中の最大・最小画素値を調べる関数等
- **コピー関数**
  画像全体または指定した範囲をコピーする関数等
- **スケーリング関数**
  画像全体または指定した範囲を拡大・縮小する関数等

- **正規化関数**
  画像全体または指定した範囲を正規化する関数等
- **射影変換関数**
  射影変換行列により画像を変換する関数
- **テクスチャマッピング関数**
  テクスチャマッピングを行う関数
- **XY 画像に関する関数**
  XY 画像とは光軸点に原点を持ち，垂直方向上方を X 軸，垂直方向右方を Y 軸とした座標系を持つ画像です．
  XY 画像を扱うための関数は，TeoOpenXYFile, TeoCreateXYFile のように libteo が提供する関数に XY が挿入された形の関数名を持ちます．その他にも TEO 画像の各パラメータにアクセスするために用意されたマクロと同じ働きをするマクロも用意されています．

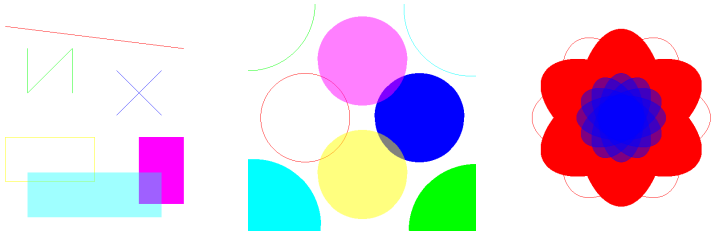
D.5　libteo_draw

libteo_draw は TEO 画像上に線分や○印，□印等の図形を描画するための拡張ライブラリです．画像解析を行う上では必要のないライブラリですが，解析結果をわかりやすく表示するための支援として有効です．

libteo_draw を使うと以下に示す図形を描画することができます．

- **点** ... TeoDrawPoint
- **線分** ... TeoDrawLine
- **連続した線分** ... TeoDrawLines
- **複数の線分** ... TeoDrawSegments
- **矩形** ... TeoDrawRectangle
- **凸形** ... TeoDrawConvex
- **円形** ... TeoDrawArc
- **楕円** ... TeoDrawEllipse
- **任意の 2 次曲線** ... TeoDrawConic

libteo_draw によって描画した図形の例を図 D.1に示します．



(a) 線分，矩形の描画　(b) 円の描画　(c) 楕円の描画

図 D.1: libteo_draw を用いた図形の描画

D.6 libteo2gdk-pixbuf

libteo2gdk-pixbuf は TEO 画像データを GdkPixbuf 形式のデータに変換するライブラリです．GdkPixbuf は GTK+ ライブラリ内で扱われる画像データ形式であり，GTK+ を利用した GUI ツールを作成する際に役にたちます．

TEO 画像データから GdkPixbuf 形式に変換するだけではなく，反対に GdkPixbuf 形式から TEOIMAGE 形式への変換も行うことができます．これにより，gdk-pixbuf 関数により読み込んだ JPEG, PNG 等の画像データを TEO 画像データに変換することが可能になります．

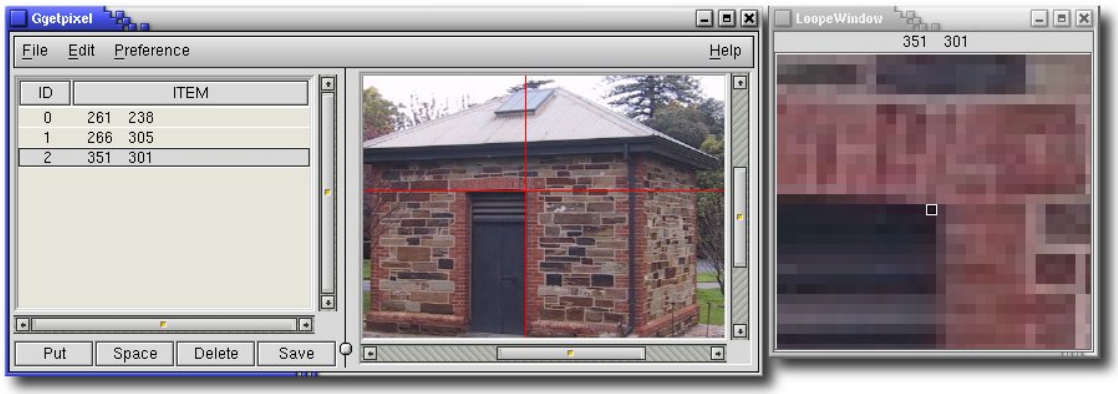


図 D.2: libteo2gdk-pixbuf を利用した GUI ツール (ggetpixel)

付録 E

MMX 命令を使用した高速処理

ここでは，MMX 命令を用いた画像処理の高速化について解説します．MMX 命令に関する資料[*1]，MMX 命令を用いた画像処理の例が少ないことから，ここではそれほど詳しい説明はできません．もし興味を持ったら詳細を調べてより詳しい資料を書いてもらえると幸いです．

MMX 命令を使用する利点は単純にいうと複数の画素を一度に処理できることです．MMX 命令では 64 ビットレジスタに格納した複数の画素データを一度に扱うことができるため，8 ビットの画像データであれば一度に 8 画素を処理することができます．単純に計算するとこれだけで処理時間が MMX 命令を使用しない場合と比較して 1/8 に短縮されるわけです．

MMX 命令による加算の例を図 E.1に示します．図 E.1(a) はラップラウンドの加算です．ラップラウンドで計算するとオバーフロー，アンダーフローした分は無視されます．また飽和モードでの計算では，オバーフロー，アンダーフローを起こした場合，上限または下限に飽和します (図 E.1(b))．MMX 命令一覧を表 E.1に掲載しておきます．命令の詳細は各自で調べて下さい．

表 E.1を見ておわかりのように MMX 命令はアセンブラ命令ですので，MMX 命令を使ったプログラムを書くためにはソースをアセンブラで記述しなければなりません．しかし C 言語のソース中にアセンブラのコードを挿入するインラインアセンブラの機能を使用してプログラムを書くことができます．これによってアセンブラで記述するには面倒な部分は C 言語で記述し，高速処理したい画像処理部分をアセンブラで記述することができます．

ここでは背景差分によって運動領域を検出するプログラムを例に，C 言語のソース中で MMX 命令を使用す

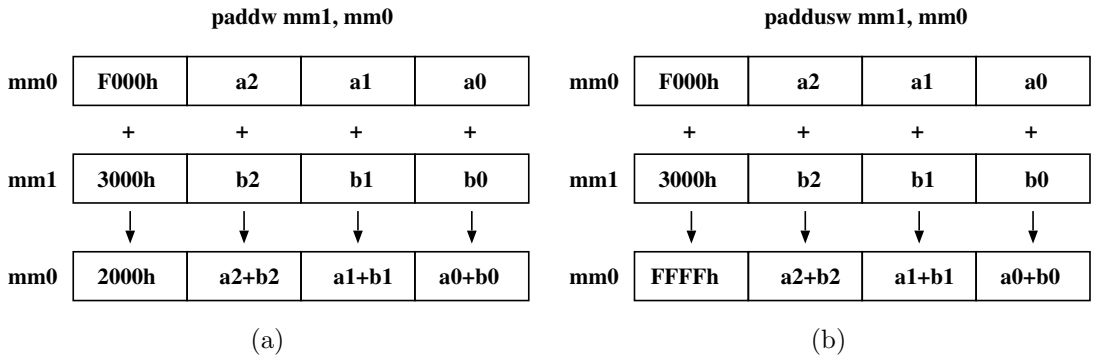


図 E.1: MMX 命令による加算の例 (a) ラップラウンド (b) 符号なし・飽和

[*1] 「MMX テクノロジ最適化テクニック」著者：小鷲英一　発行所：株式会社アスキーは MMX を勉強するのにお勧めです．

る方法について説明します．リスト E.1 にソースを示します．メイン関数についてはいまさら説明することもないと思いますので，MMX 命令を使用する部分について詳しく説明します．図 E.2 に MMX レジスタの値の変化を示します．

変数の宣言 (14 行目)　運動領域を検出するためのしきい値の宣言と初期化．MMX 命令で使用するために 8 画素分の値を用意しています．

ポインタの初期化 (16–18 行目)　入力画像データ，出力画像データのアドレスを別のポインタに代入します．

画像処理ループ (20–33 行目)　画像データにアクセスするためのループ．1 回のループで 8 画素分処理するので n の増分 (減分?) は 8 です．

入力画像データ 1 からのデータ取得 (21 行目)　*src_ptr*1 の指すデータを MMX 用のレジスタ mm0 に代入します．インラインアセンブラは次のフォーマット[*2]で記述します．

asm (”operand source, destination”);　or　**asm (”operand source”);**

特殊なケースとして変数の値をレジスタに代入したりレジスタの値を変数に代入する場合があります．この命令では，変数 *src_ptr*1 の値を 64 ビット単位でレジスタ mm0 に代入しています．

この場合次のようなフォーマットを用います．

asm (”operand (%0), destination”::”制約[*3] **”(変数));**

MMX レジスタとして使用できるのは mm0 から mm7 の 8 つのレジスタです．

入力画像データ 1 からのデータ取得 (22 行目)　*src_ptr*2 の指すデータを MMX 用のレジスタ mm1 に代入します．

値のコピー (23 行目)　mm0 レジスタの内容を mm2 レジスタにコピーします．

差分処理 1(24 行目)　レジスタ mm0 とレジスタ mm1 の内容の差分を取り，結果をレジスタ mm0 に代入します．psubusb は飽和・符号なしのバイト単位の減算命令を表します．飽和とは演算結果がアンダーフローした場合は下限値，オーバーフローした場合は上限値とするものです．この場合，差が正値だった時のみ差の値が入り，負値だった場合 0 が入ります．

差分処理 (25 行目)　レジスタ mm1 とレジスタ mm2 の内容の差分を取り，結果をレジスタ mm1 に代入します

差の絶対値を計算 (26 行目)　両方の差分値 mm0, mm1 の or を取ります．これにより，レジスタ mm0 に画像データ 1 と画像データ 2 の差の絶対値が格納されます．

しきい値処理 (27 行目)　差の絶対値としきい値を比較して，差の絶対値がしきい値より大きい場合は 0FFh，そうでなければ 0 をレジスタ mm0 に格納します．

出力値の計算 (28 行目)　上の結果とレジスタ mm2(画像データ 2 の値が格納されている) との AND を取ります．これにより差分値がしきい値より大きい画素には画像データ 2 の画素値が，そうでない場合は 0 が格納されます．

結果を出力データにコピー (29 行目)　処理結果を出力画像データにコピーします．

MMX レジスタの値を変数にコピーするには次のようなフォーマットを用います．

asm (”operand source, (%0)”::”制約”(変数));

アクセス画素の移動 (30–32 行目)　画像ポインタを 8 画素分移動します．

後始末 (34 行目)　MMX 命令を使った後のおまじないだと思いましょう．

[*2] 詳細は http://www.sra.co.jp/wingnut/gcc/gcc-j.html#Extended%20Asm を参照して下さい．

[*3] 詳細は http://www.sra.co.jp/wingnut/gcc/gcc-j.html#Constraints を参照下さい．

リスト E.1: MMX 命令を用いた背景差分

```
/* ************************************************************ mmx.c *** */
#include <stdio.h>
#include <stdlib.h>
#include <teo.h>

/* 背景差分関数（MMX 命令使用）*********************************************** */
static void
func_subtract_mmx (unsigned char        *src1,
                   unsigned char        *src2,
                   unsigned char        *dst,
                   int                  size) {
  int           n;
  unsigned char        *src_ptr1, *src_ptr2, *dst_ptr;
  unsigned char threshold[8] = {16, 16, 16, 16, 16, 16, 16, 16};

  src_ptr1 = src1;
  src_ptr2 = src2;
  dst_ptr  = dst;

  for (n = size; n > 0; n -= 8) {
    asm ("movq (%0), %%mm0"::"r"(src_ptr1));
    asm ("movq (%0), %%mm1"::"r"(src_ptr2));
    asm ("movq %mm0, %mm2");
    asm ("psubusb %mm1, %mm0");
    asm ("psubusb %mm2, %mm1");
    asm ("por %mm1, %mm0");
    asm ("pcmpgtb (%0), %%mm0"::"r"(threshold));
    asm ("pand %mm2, %mm0");
    asm ("movq %%mm0, (%0)"::"r"(dst_ptr));
    src_ptr1 += 8;
    src_ptr2 += 8;
    dst_ptr  += 8;
  }
  asm ("emms");
}

/* メイン関数 ***************************************************************** */
int
```

```
39 main (int       argc,
40       char      **argv) {
41   TEOFILE       *src_teofp[2], *dst_teofp;
42   TEOIMAGE      *src_img[2], *dst_img;
43   int           n;
44
45   /* 入力画像の読み込み */
46   for (n = 0; n < 2; n++) {
47     src_teofp[n] = TeoOpenFile (argv[n+1]);
48     src_img[n]   = TeoAllocSimilarImage (src_teofp[n]);
49     TeoReadFrame (src_teofp[n], src_img[n]);
50   }
51   /* 出力画像の準備 */
52   dst_teofp = TeoCreateSimilarFile ("-", src_teofp[0]);
53   dst_img   = TeoAllocSimilarImage (dst_teofp);
54
55   /* 背景差分 */
56   func_subtract_mmx ((unsigned char *) TeoData (src_img[1]),
57                      (unsigned char *) TeoData (src_img[0]),
58                      (unsigned char *) TeoData (dst_img),
59                      TeoFsize (src_teofp[0]));
60
61   TeoWriteFrame (dst_teofp, dst_img);
62   TeoCloseFile (src_teofp[0]);
63   TeoCloseFile (src_teofp[1]);
64   TeoCloseFile (dst_teofp);
65   TeoFreeImage (src_img[0]);
66   TeoFreeImage (src_img[1]);
67   TeoFreeImage (dst_img);
68
69   return 0;
70 }
71
72 /* ********************************************************* End of mmx.c *** */
```

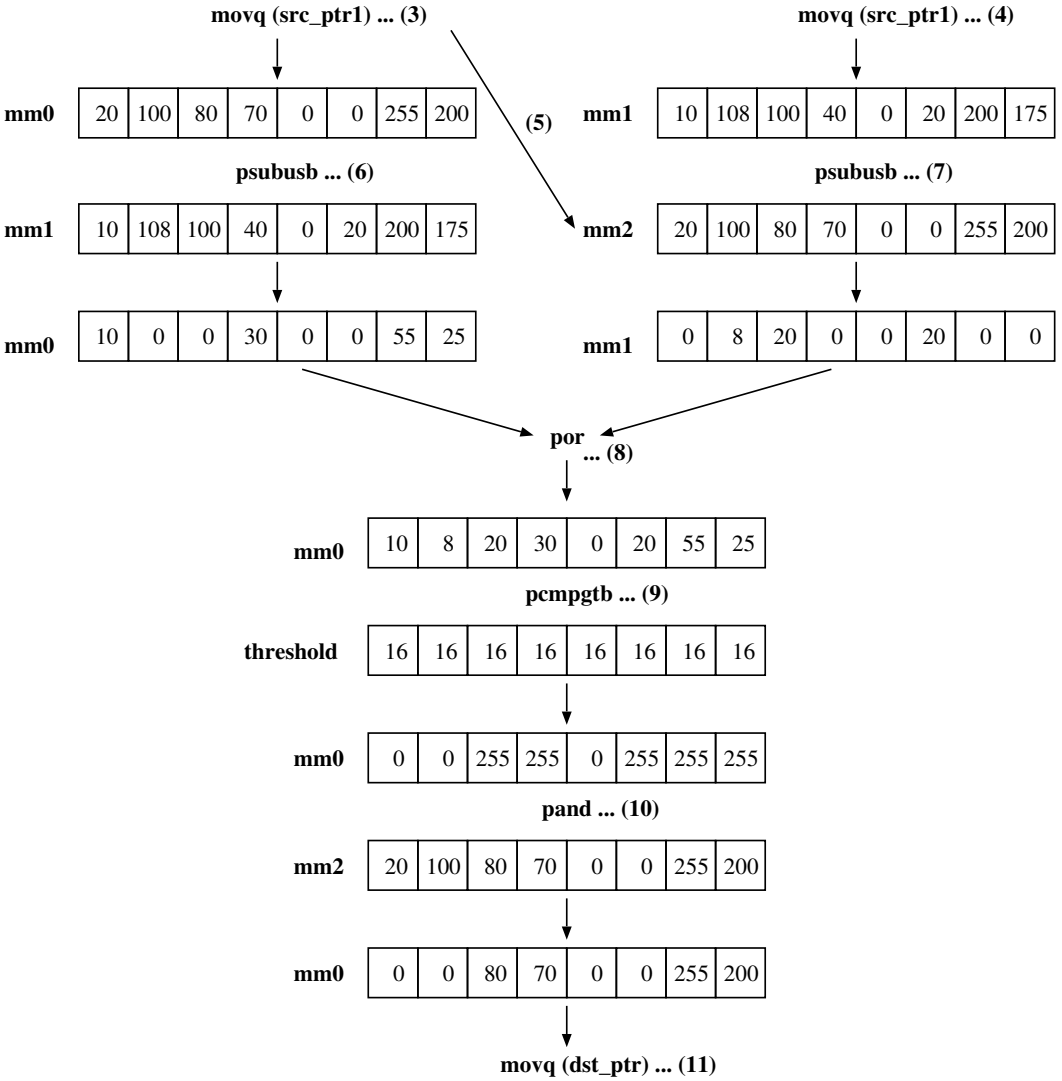


図 E.2: MMX 命令によるレジスタ値の変化

(a) (b) (c)

図 E.3: MMX 命令を用いた背景差分 (a) 背景画像 (b) 入力画像 (c) 検出された運動領域

表 E.1: MMX 命令一覧

命令	ラップラウンド				符号付き飽和				符号なし飽和			
	b	w	d	q	b	w	d	q	b	w	d	q
算術												
padd	**b**	**w**	**d**		**sb**	**sw**			**usb**	**usw**		
psub	**b**	**w**	**d**		**sb**	**sw**			**usb**	**usw**		
pmul(l|h)		**w**										
pmadd		**wd** →										
論理												
pand					×							
pandn					×							
por					×							
pxor					×							
シフト												
ps(ll|rl)		**w**	**d**	**q**								
psra		**w**	**d**									
比較												
pcmp(eq|gt)	**b**	**w**	**d**									
変換												
pack					**sswb** ←		**ssdw** ←		**uswb** ←			
punpack(l|h)	**bw** →		**wd** →	**dq** →								
移動												
mov			**d**	**q**								
emms												

索引

かいていだい3はん テオライブラリによるがぞうしょりプログラミングにゅうもん
［改定第3版］ TEOライブラリによる画像処理プログラミング入門

2002 年　3 月　22 日　初　版
2003 年　5 月　 9 日　第 2 版
2004 年　4 月　 1 日　第 3 版
著者　　菅谷保之
E-mail　sugaya@suri.it.okayama-u.ac.jp